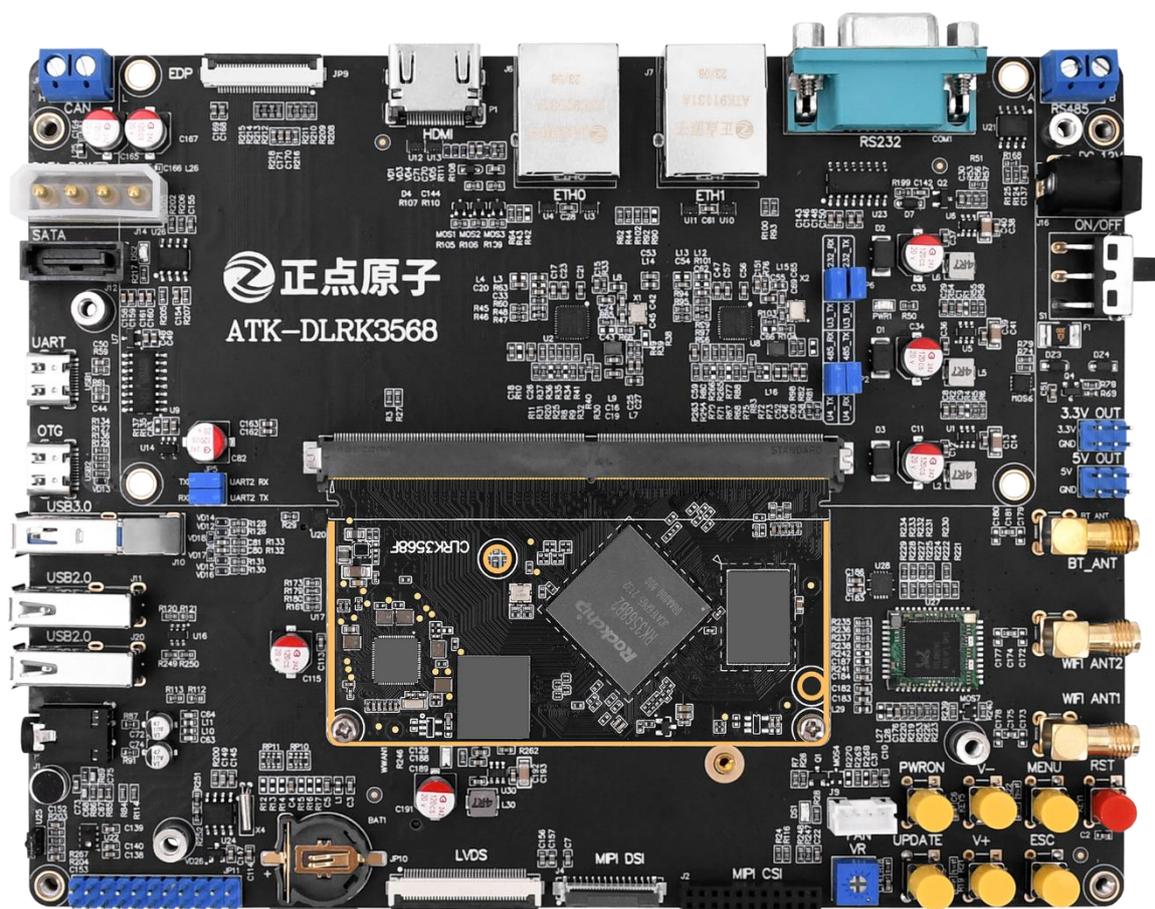


ATK-DLRK3568 嵌入式 Linux 驱动开发指南 V1.0

-正点原子 ATK-DLRK3568





正点原子公司名称 : 广州市星翼电子科技有限公司

原子哥在线教学平台 : www.yuanzige.com

开源电子网 / 论坛 : <http://www.openedv.com/forum.php>

正点原子淘宝店铺 : <https://openedv.taobao.com>

正点原子官方网站 : www.alientek.com

正点原子 B 站视频 :

<https://space.bilibili.com/394620890>

电话: 020-38271790 传真: 020-36773971

请关注正点原子公众号, 资料发布更新我们会通知。

请下载原子哥 APP, 数千讲视频免费学习, 更快更流畅。



扫码关注正点原子公众号



扫码下载“原子哥”APP

文档更新说明

版本	版本更新说明	负责人	校审	发布日期
V1.0	初稿:	正点原子 linux 团队	正点原子 linux 团队	2023.07.03

免责声明

本档所提及的产品规格和使用说明仅供参考，如有内容更新，恕不另行通知；除非有特殊约定，本档仅作为产品指导，所作陈述均不构成任何形式的担保。本档版权归广州市星翼电子科技有限公司所有，未经公司的书面许可，任何单位和个人不得以营利为目的进行任何方式的传播。

为了得到最新版本的产品信息，请用户定时访问正点原子资料下载中心或者与淘宝正点原子旗舰店客服联系索取。感谢您的包容与支持。

正点原子资料下载中心: <http://www.openedv.com/docs/index.html>

目录

免责声明.....	4
第一章 开发环境搭建.....	14
第二章 SDK 包的使用.....	15
第三章 U-Boot 使用.....	16
3.1 U-Boot 简介.....	17
3.2 U-Boot 初次编译.....	18
3.2.1 编译与烧写.....	18
3.2.2 U-Boot 启动过程.....	18
3.3 U-Boot 命令使用.....	20
3.3.1 查询命令.....	21
3.3.2 环境变量操作命令.....	22
3.3.3 内存操作命令.....	23
3.3.4 网络操作命令.....	26
3.3.5 EMMC 和 SD 卡操作命令.....	33
3.3.6 EXT 格式文件系统操作命令.....	38
3.3.7 BOOT 操作命令.....	39
3.3.8 其他常用命令.....	42
3.3.9 MII 命令使用说明.....	43
第四章 Linux 驱动开发准备工作.....	45
4.1 Linux 内核编译.....	46
4.1.1 编译 Linux 内核.....	46
4.1.2 关闭内核 log 时间戳.....	46
4.2 根文件系统确认.....	47
4.2.1 创建/lib/modules/4.19.232 目录.....	47
4.2.2 检查相关命令是否存在.....	47
4.3 Ubuntu 下使用 ADB 向开发板发送文件.....	47
4.4 安装驱动开发所使用的交叉编译器.....	48
第五章 字符设备驱动开发.....	51
5.1 字符设备驱动简介.....	52
5.2 字符设备驱动开发步骤.....	55
5.2.1 驱动模块的加载和卸载.....	55
5.2.2 字符设备注册与注销.....	56
5.2.3 实现设备的具体操作函数.....	58
5.2.4 添加 LICENSE 和作者信息.....	60
5.3 Linux 设备号.....	61
5.3.1 设备号的组成.....	61
5.3.2 设备号的分配.....	61
5.4 chrdevbase 字符设备驱动开发实验.....	62

5.4.1 实验程序编写.....	62
5.4.2 编写测试 APP	70
5.4.3 编译驱动程序和测试 APP	74
5.4.4 运行测试.....	75
第六章 嵌入式 Linux LED 驱动开发实验.....	79
6.1 Linux 下 LED 灯驱动原理	80
6.1.1 地址映射.....	80
6.1.2 I/O 内存访问函数.....	82
6.2 硬件原理图分析.....	82
6.3、RK3568 GPIO 驱动原理讲解	83
6.3.1 引脚复用设置.....	83
6.3.2 引脚驱动能力设置.....	85
6.3.3 GPIO 输入输出设置.....	86
6.3.4 GPIO 引脚高低电平设置.....	87
6.4 实验程序编写.....	87
6.4.1 LED 灯驱动程序编写.....	88
6.4.2 编写测试 APP	94
6.5 运行测试.....	96
6.5.1 编译驱动程序和测试 APP	96
6.5.2 运行测试.....	96
第七章 新字符设备驱动实验	98
7.1 新字符设备驱动原理.....	99
7.1.1 分配和释放设备号.....	99
7.1.2 新的字符设备注册方法.....	100
7.2 自动创建设备节点.....	101
7.2.1 mdev 机制.....	101
7.2.1 创建和删除类.....	102
7.2.2 创建设备.....	102
7.2.3 参考示例.....	103
7.3 设置文件私有数据.....	103
7.4 硬件原理图分析.....	104
7.5 实验程序编写.....	104
7.5.1 LED 灯驱动程序编写.....	104
7.5.2 编写测试 APP	112
7.6 运行测试.....	112
7.6.1 编译驱动程序和测试 APP	112
7.6.2 运行测试.....	113
第八章 Linux 设备树.....	114
8.1 什么是设备树?	115
8.2 DTS、DTB 和 DTC.....	116
8.3 DTS 语法.....	118

8.3.1 .dtsi 头文件.....	118
8.3.2 设备节点.....	121
8.3.3 标准属性.....	124
8.3.4 根节点 compatible 属性.....	128
8.3.5 向节点追加或修改内容.....	129
8.4 创建小型模板设备树.....	130
8.5 设备树在系统中的体现.....	133
8.6 特殊节点.....	134
8.6.1 aliases 子节点.....	134
8.6.2 chosen 子节点.....	135
8.7 绑定信息文档.....	136
8.8 设备树常用 OF 操作函数.....	138
8.8.1 查找节点的 OF 函数.....	138
8.8.2 查找父/子节点的 OF 函数.....	140
8.8.3 提取属性值的 OF 函数.....	141
8.8.4 其他常用的 OF 函数.....	144
第九章 设备树下的 LED 驱动实验.....	147
9.1 设备树 LED 驱动原理.....	148
9.2 硬件原理图分析.....	148
9.3 实验程序编写.....	148
9.3.1 修改设备树文件.....	148
9.3.2 LED 灯驱动程序编写.....	150
9.3.3 编写测试 APP.....	158
9.4.1 编译驱动程序和测试 APP.....	158
9.4.2 运行测试.....	159
第十章 pinctrl 和 gpio 子系统实验.....	160
10.1 pinctrl 子系统.....	161
10.1.1 pinctrl 子系统简介.....	161
10.1.2 rk3568 的 pinctrl 子系统驱动.....	161
10.1.3 设备树中添加 pinctrl 节点模板.....	171
10.2 gpio 子系统.....	172
10.2.1 gpio 子系统简介.....	172
10.2.2 rk3568 的 gpio 子系统驱动.....	172
10.2.3 gpio 子系统 API 函数.....	174
10.2.4 设备树中添加 gpio 节点模板.....	175
10.2.5 与 gpio 相关的 OF 函数.....	175
10.3 硬件原理图分析.....	176
10.4 实验程序编写.....	176
10.4.1 检查 IO 是否已经被使用.....	176
10.4.2 修改设备树文件.....	177
10.4.3 LED 灯驱动程序编写.....	178
10.4.4 编写测试 APP.....	185

10.5 运行测试.....	185
10.5.1 编译驱动程序和测试 APP	185
10.5.2 运行测试.....	185
第十一章 Linux 并发与竞争.....	187
11.1 并发与竞争.....	188
11.2 原子操作.....	189
11.2.1 原子操作简介.....	189
11.2.2 原子整形操作 API 函数	190
11.2.3 原子位操作 API 函数	191
11.3 自旋锁.....	192
11.3.1 自旋锁简介.....	192
11.3.2 自旋锁 API 函数	193
11.3.3 其他类型的锁.....	195
11.3.4 自旋锁使用注意事项.....	196
11.4 信号量.....	197
11.4.1 信号量简介.....	197
11.4.2 信号量 API 函数	198
11.5 互斥体.....	198
11.5.1 互斥体简介.....	198
11.5.2 互斥体 API 函数	199
第十二章 Linux 并发与竞争实验.....	200
12.1 原子操作实验.....	201
12.1.1 实验程序编写.....	201
12.1.2 运行测试.....	210
12.2 自旋锁实验	211
12.2.1 实验程序编写.....	211
12.2.2 运行测试.....	219
12.3 信号量实验.....	219
12.3.1 实验程序编写.....	220
12.3.2 运行测试.....	223
12.4 互斥体实验.....	224
12.4.1 实验程序编写.....	224
12.4.2 运行测试.....	227
第十三章 Linux 按键输入实验.....	229
13.1 Linux 下按键驱动原理	230
13.2 硬件原理图分析.....	230
13.3 实验程序编写.....	230
13.3.1 修改设备树文件.....	230
13.3.2 按键驱动程序编写.....	231
13.3.3 编写测试 APP	238
13.4 运行测试.....	240

13.4.1 编译驱动程序和测试 APP	240
13.4.2 运行测试.....	240
第十四章 Linux 内核定时器实验.....	242
14.1 Linux 时间管理和内核定时器简介.....	243
14.1.1 内核时间管理简介.....	243
14.1.2 内核定时器简介.....	246
14.1.3 Linux 内核短延时函数	248
14.2 硬件原理图分析.....	249
14.3 实验程序编写.....	249
14.3.1 修改设备树文件.....	249
14.3.2 定时器驱动程序编写.....	249
14.3.3 编写测试 APP	257
14.4 运行测试.....	259
14.4.1 编译驱动程序和测试 APP	259
14.4.2 运行测试.....	260
第十五章 Linux 中断实验.....	261
15.1 Linux 中断简介	262
15.1.1 Linux 中断 API 函数.....	262
15.1.2 上半部与下半部.....	264
15.1.3 设备树中断信息节点.....	270
15.1.4 获取中断号.....	275
15.2 硬件原理图分析.....	275
15.3 实验程序编写.....	276
15.3.1 修改设备树文件.....	276
15.3.2 按键中断驱动程序编写.....	277
15.3.2 编写测试 APP	285
15.4 运行测试.....	287
15.4.1 编译驱动程序和测试 APP	287
15.4.2 运行测试.....	288
第十六章 Linux 阻塞和非阻塞 IO 实验.....	289
16.1 阻塞和非阻塞 IO	290
16.1.1 阻塞和非阻塞简介.....	290
16.1.2 等待队列.....	291
16.1.3 轮询	293
16.1.4 Linux 驱动下的 poll 操作函数	297
16.2 阻塞 IO 实验.....	298
16.2.1 硬件原理图分析.....	298
16.2.2 实验程序编写.....	298
16.2.3 运行测试.....	301
16.3 非阻塞 IO 实验.....	303
16.3.1 硬件原理图分析.....	303

16.3.2 实验程序编写.....	303
16.3.3 运行测试.....	307
第十七章 异步通知实验.....	309
17.1 异步通知.....	310
17.1.1 异步通知简介.....	310
17.1.2 驱动中的信号处理.....	312
17.1.3 应用程序对异步通知的处理.....	314
17.2 硬件原理图分析.....	314
17.3 实验程序编写.....	314
17.3.1 修改设备树文件.....	315
17.3.2 程序编写.....	315
17.3.3 编写测试 APP.....	317
17.4 运行测试.....	319
17.4.1 编译驱动程序和测试 APP.....	319
17.4.2 运行测试.....	320
第十八章 platform 设备驱动实验.....	321
18.1 Linux 驱动的分离与分层.....	322
18.1.1 驱动的分隔与分离.....	322
18.1.2 驱动的分层.....	324
18.2 platform 平台驱动模型简介.....	324
18.2.1 platform 总线.....	324
18.2.2 platform 驱动.....	326
18.2.3 platform 设备.....	331
18.3 硬件原理图分析.....	334
18.4 试验程序编写.....	334
18.4.1 platform 设备与驱动程序编写.....	334
18.4.2 测试 APP 编写.....	344
18.5 运行测试.....	346
18.5.1 编译驱动程序和测试 APP.....	346
18.4.2 运行测试.....	346
第十九章 设备树下的 platform 驱动编写.....	348
19.1 设备树下的 platform 驱动简介.....	349
19.1.1 创建设备的 pinctrl 节点.....	349
19.1.2 在设备树中创建设备节点.....	349
19.1.3 编写 platform 驱动的时候要注意兼容属性.....	350
19.2 检查引脚复用配置.....	350
19.2.1 检查引脚 pinctrl 配置.....	350
19.2.2 检查 GPIO 占用.....	352
19.3 硬件原理图分析.....	352
19.4 实验程序编写.....	352
19.4.1 修改设备树文件.....	352

19.4.2 platform 驱动程序编写.....	352
19.4.3 编写测试 APP	359
19.5 运行测试.....	359
19.5.1 编译驱动程序和测试 APP	359
19.5.2 运行测试.....	359
第二十章 Linux 自带的 LED 灯驱动实验.....	361
20.1 Linux 内核自带 LED 驱动使能	362
20.2 Linux 内核自带 LED 驱动简介	363
20.2.1 LED 灯驱动框架分析.....	363
20.2.2 module_platform_driver 函数简析.....	365
20.2.3 gpio_led_probe 函数简析	366
20.3 设备树节点编写.....	368
20.4 运行测试.....	369
第二十一章 Linux MISC 驱动实验	371
21.1 MISC 设备驱动简介.....	372
21.2 硬件原理图分析.....	373
21.3 实验程序编写.....	373
21.3.1 修改设备树.....	373
21.3.2 misc 驱动程序编写	374
21.3.3 编写测试 APP	379
21.4 运行测试.....	381
21.4.1 编译驱动程序和测试 APP	381
21.4.2 运行测试.....	381
第二十二章 Linux INPUT 子系统实验.....	383
22.1 input 子系统	384
22.1.1 input 子系统简介	384
22.1.2 input 驱动编写流程	384
22.1.3 input_event 结构体	390
22.2 硬件原理图分析.....	391
22.3 实验程序编写.....	391
22.3.1 修改设备树文件.....	391
22.3.2 按键 input 驱动程序编写	391
22.3.3 编写测试 APP	397
22.4 运行测试.....	399
22.4.1 编译驱动程序和测试 APP	399
22.4.2 运行测试.....	399
22.5 Linux 自带按键驱动程序的使用.....	401
22.5.1 使能内核自带按键驱动程序源码简析.....	401
22.5.2 自带 ADC 按键驱动程序的使用	402
第二十三章 Linux PWM 驱动实验.....	405

23.1 PWM 驱动简析.....	406
23.1.1 设备树下的 PWM 控制器节点.....	406
23.1.2 PWM 子系统.....	408
23.1.3 PWM 驱动源码分析.....	409
23.2 PWM 驱动编写.....	413
23.2.1 修改设备树.....	413
23.2.2 使能 PWM 驱动.....	415
23.3 PWM 驱动测试.....	415
第二十四章 MIPI DSI 屏幕驱动实验.....	418
24.1 MIPI 联盟简介.....	419
24.2 MIPI DSI 概述.....	424
24.2.1 MIPI DSI 协议综述.....	424
24.2.2 MIPI DSI 分层.....	425
24.3 MIPI DSI 物理层和 D-PHY.....	429
24.3.1 什么是 Lane.....	430
24.3.2 D-PHY 信号电平.....	431
24.3.3 通道状态.....	431
24.3.4 数据 Lane 三种工作模式.....	432
24.5 video 和 command 模式.....	434
24.5.1 command 模式.....	434
24.5.2 video 模式.....	435
24.6 DBI 和 DPI 格式.....	436
24.6.1 DBI 接口.....	436
24.6.2 DPI 接口.....	437
24.7 video 模式下三种传输方式.....	440
24.7.1 Non-Burst Mode with Sync Pluses.....	440
24.7.2 Non-Burst Mode with Sync Events.....	441
24.7.3 Burst Mode.....	442
24.8 长短数据包.....	442
24.8.1 数据包概述.....	442
24.8.2 数据包字节排序策略.....	443
24.8.3 长数据包.....	443
24.8.4 短数据包.....	444
24.9 指令类型.....	445
24.10 MIPI DSI 时钟计算.....	448
24.11 RK3568 MIPI DSI 介绍.....	449
24.12 硬件原理图分析.....	450
24.13 MIPI 屏幕驱动调试思路.....	451
24.14 实验程序编写.....	452
24.14.1 背光驱动.....	452
24.14.2 屏幕初始化序列发送时序参数设置.....	454
24.14.3 DSI 设备树节点编写.....	456

24.15 运行测试.....	461
24.15.1 LCD 背光调节	461
24.15.2 MIPI 协议实测	462
第二十五章 HDMI 屏幕驱动实验.....	467
第二十六章 LVDS 屏幕驱动实验	467
第二十七章 EDP 屏幕驱动实验.....	467
第二十八章 Linux I2C 驱动实验.....	468
28.1 I2C& AP3216C 简介	469
28.1.1 I2C 简介	469
28.1.2 RK3568 I2C 简介	471
28.1.3 AP3216C 简介	472
28.2 Linux I2C 总线框架简介	473
28.2.1 I2C 总线驱动	474
28.2.2 I2C 总线设备	477
28.2.3 I2C 设备和驱动匹配过程.....	481
28.3 RK3568 I2C 适配器驱动分析.....	482
28.4 I2C 设备驱动编写流程.....	489
28.4.1 I2C 设备信息描述.....	489
28.4.2 I2C 设备数据收发处理流程.....	491
28.5 硬件原理图分析.....	494
28.6 实验程序编写.....	495
28.6.1 修改设备树.....	495
28.6.2 AP3216C 驱动编写	496
28.6.3 编写测试 APP	507
28.7 运行测试.....	508
28.7.1 编译驱动程序和测试 APP	508
28.7.2 运行测试.....	509
第二十九章 Linux RTC 驱动实验.....	510
29.1 Linux 内核 RTC 驱动简介	511
29.2 ATK-DLRK3568 核心板 RTC 驱动分析	515
29.3 RTC 时间查看与设置.....	523
29.3.1 使能 RK809 内部 RTC	523
29.3.2 查看时间.....	525

第一章 开发环境搭建

开发环境搭建请参考资料：[开发板光盘](#)→10、[用户手册](#)→02 开发文档→【正点原子】ATK-DLRK3568 嵌入式 Linux 系统开发手册.pdf 中的第一到二章节。请用户自行安装 Ubuntu，配置好开发环境。

第二章 SDK 包的使用

传统的嵌入式 Linux 驱动开发主要就是三大巨头: uboot、Linux kernel 和根文件系统(rootfs), 比如我们已经推出的 I.MX6U 和 STM32MP157 这两款开发板。I.MX6U 和 STM32MP157 这两款芯片都比较简单, 算是很基础的 Cortex-A 内核 MPU, 所以只需要搞定三巨头就可以。但是随着芯片性能的不不断提升, 比如加入硬件视频编解码、NPU、GPU、ISP 等外设以后, 单纯的依靠三巨头就不能完全发挥出芯片的性能。因为不同的半导体厂商其外设实现不同, 因此对应的驱动也不同, 比如硬件视频编解码, 每个厂商都有自己的驱动库, 并且他们会向用户提供一套 API。用户要使用这些 API 编写自己的应用程序, 就需要链接到这些库, 而且也可能需要其他的第三方库支持。总之就是各种交织, 各种依赖, 已经不单单是那三巨头的的事情了。如果要靠用户去解决这些依赖是不现实的, 所以半导体厂商会将这些东西打包到一起提供给用户, 也就是 SDK 包(提供的 SDK 包有两种, 一种是安卓 SDK, 另一种是 Linux SDK)。本教程是 Linux 驱动开发, 所以我们在这个文档是基于 Linux SDK 开发。

以我们使用的 ATK-DLRK3568 开发板使用的 RK3568 芯片为例, 瑞芯微提供了全面的 Linux SDK 包, Linux SDK 包主要包含了:

- app: 存放上层应用 app, 包括 Qt 应用程序, 以及其它的 C/C++ 应用程序。
- buildroot: 基于 buildroot 开发的根文件系统。
- debian: 基于 Debian 开发的根文件系统。
- device/rockchip: 存放各芯片板级配置文件和 Parameter 文件, 以及一些编译与打包固件的脚本和预备文件。
- docs: 存放芯片模块开发指导文档、平台支持列表、芯片平台相关文档、Linux 开发指南等。
- external: 存放所需的第三方库, 包括音频、视频、网络、recovery 等。
- kernel: Linux 4.19 版本内核源码。
- prebuilts: 存放交叉编译工具链。
- rkbin: 存放 Rockchip 相关的 Binary 和工具。
- rockdev: 存放编译输出固件, 编译 SDK 后才会生成该文件夹。
- tools: 存放 Linux 和 Windows 操作系统环境下常用的工具, 包括镜像烧录工具、SD 卡升级启动制作工具、批量烧录工具等, 譬如前面给大家介绍的 RKDevTool 工具以及 Linux_Upgrade_Tool 工具在该目录下均可找到。
- u-boot: 基于 v2017.09 版本进行开发的 uboot 源码。
- yocto: 基于 Yocto 开发的根文件系统。

所以本教程, 包括我们其他的 RK3568 Linux 教程资料都是跟这个 LinuxSDK 包打交道。

关于 SDK 包更加详细的介绍请参考资料: [开发板光盘](#)→10、[用户手册](#)→【正点原子】ATK-DLRK3568 嵌入式 Linux 统开发手册.pdf 中的第四章。

第三章 U-Boot 使用

U-Boot 用于引导 Linux 系统,所以要先使用一下 U-Boot,体验一下 U-Boot 是个什么东西。正点原子 ATK-DLRK3568 开发板出厂系统已经烧写了 U-Boot,所以可以直接拿开发板体验 U-Boot。当然了,提供的 SDK 包里面也有 U-Boot 源码,大家也可以自行编译 U-Boot 源码烧写测试。本章我们主要学习使用 U-Boot 的相关命令。

3.1 U-Boot 简介

Linux 系统要启动需要通过 bootloader 程序引导, 也就是说芯片上电以后先运行一段 bootloader 程序。这段 bootloader 程序会先初始化 DDR 等外设, 然后将 Linux 内核从 flash(NAND, NOR FLASH, SD, EMMC 等)拷贝到 DDR 中, 最后启动 Linux 内核。当然了, bootloader 的实际工作要复杂的多, 但是它最主要的工作就是启动 Linux 内核, bootloader 和 Linux 内核的关系就跟 PC 上的 BIOS 和 Windows 的关系一样, bootloader 就相当于 BIOS。所以我们要先搞定 bootloader, 很庆幸, 有很多现成的 bootloader 软件可以使用, 比如 U-Boot、vivi、RedBoot 等等, 其中以 U-Boot 使用最为广泛, 为了方便书写, 本教程会将 U-Boot 写为 uboot。

uboot 的全称是 Universal Boot Loader, uboot 是一个遵循 GPL 协议的开源软件, uboot 是一个裸机代码, 可以看作是一个裸机综合例程。现在的 uboot 已经支持液晶屏、网络、USB 等高级功能。uboot 官网为 <https://www.denx.de/project/u-boot/>, 如图 3.1.1 所示:

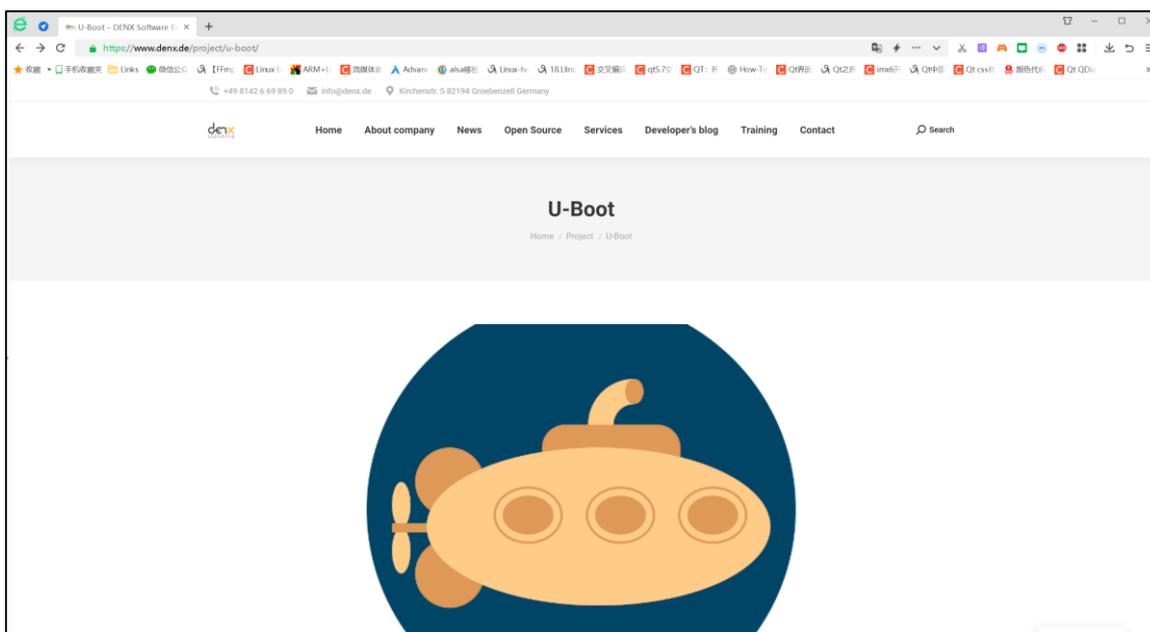


图 3.1.1 uboot 官网

我们可以在 uboot 官方 FTP 下载服务器地址为 <https://ftp.denx.de/pub/u-boot/>, 进入此 FTP 服务器即可看到 uboot 源码, 如图 3.1.2 所示:

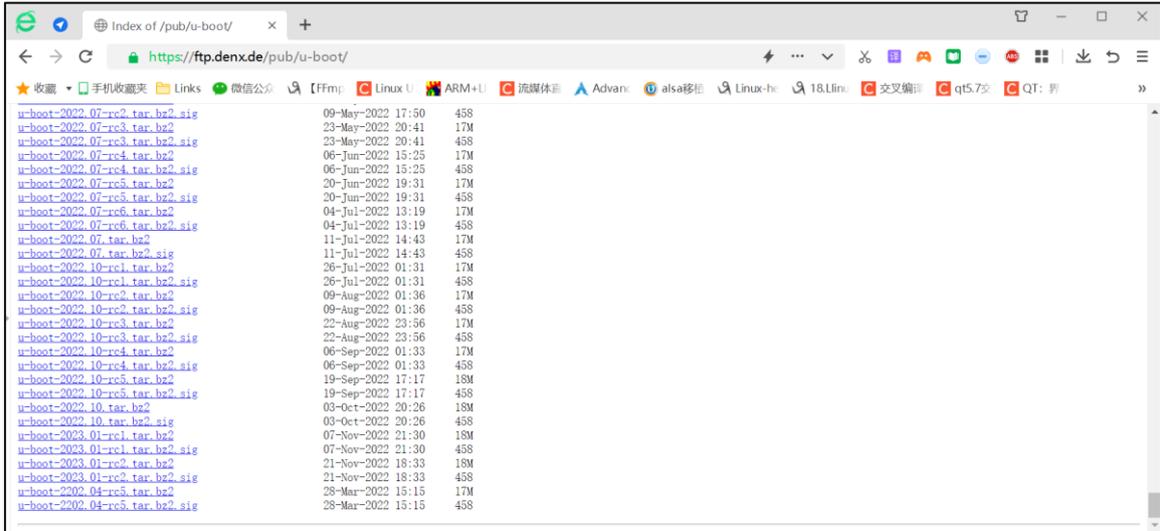


图 3.1.2 uboot 源码

图 3.1.2 中就是 uboot 原汁原味的源码文件，目前最新的版本是 2023.01-rc2。但是我们一般不会直接用 uboot 官方的 U-Boot 源码的。uboot 官方的 uboot 源码是给半导体厂商准备的，半导体厂商会下载 uboot 官方的 uboot 源码，然后将自家相应的芯片移植进去。也就是说半导体厂商会自己维护一个版本的 uboot，这个版本的 uboot 相当于是他们定制的。既然是定制的，那么肯定对自家的芯片支持会很全，虽然 uboot 官网的源码中一般也会支持他们的芯片，但是绝对是没有半导体厂商自己维护的 uboot 全面。瑞芯微的 SDK 包提供了 2017.09 版本的 uboot。

Linux SDK 包中 uboot 源码如图 3.1.3 所示：（请按[开发板光盘](#)→10、[用户手册](#)→【正点原子】[ATK-DLRK3568 嵌入式 Linux 统开发手册.pdf](#)中的第四章安装 SDK，并检索出源码）

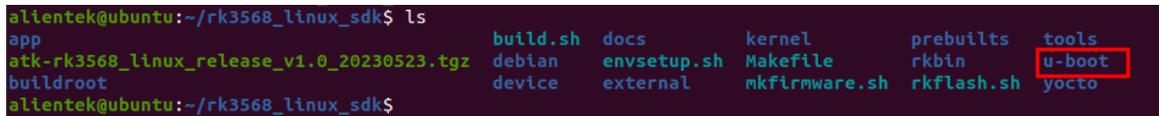


图 3.1.3 RK3568 官方 uboot 源码

图 3.1.3 中的“u-boot”就是 RK3568 官方 uboot 源码包，它支持 RK3568 芯片，而且支持各种启动方式，比如 EMMC、NAND 等等，这些都是 uboot 官方所不支持的。但是图 3.1.3 中的 uboot 是针对正点原子根据自家 ATK-DLRK3568 开发板移植好的。如果直接使用的瑞芯微原厂的 SDK 包，那么可能需要根据自己所使用的平台进行移植，使其支持自己做的板子，本教程不涉及 uboot 的移植，后面的所有实验均基于正点原子出厂 SDK 中已经移植好的 uboot。

3.2 U-Boot 初次编译

3.2.1 编译与烧写

RK3568 的 U-Boot 的编译与烧写请参考[开发板光盘](#)→10、[用户手册](#)→【正点原子】[ATK-DLRK3568 嵌入式 linux 系统开发手册.pdf](#)中的“4.4.1 单独编译 U-Boot”和“5.3.5 使用 rkflash.sh 脚本烧写（找到单独烧写 uboot 指令，自行烧写）”小节。

3.2.2 U-Boot 启动过程

烧写完成以后用 USB Type-C 线将开发板上的 UART 接口与电脑连接起来，因为我们要在串口终端里面输入命令来操作 uboot。

打开 MobaXterm, 设置好串口参数, 最后复位开发板。在 MobaXterm 上出现 “Hit key to stop autoboot(‘CTRL+C’):” 倒计时的时候按下键盘上 “CTRL+C” 按键。为了加快系统启动速度, 开发板默认是 0 秒倒计时, 所以大家一般很难卡准时间按下按键。所以建议大家按下开发板复位按键的时候, 就一直按 CTRL+C, 这样就能很轻松的进入到 uboot 的命令行。

注意! 一般 uboot 在倒计时的时候随便按下键盘上哪个按键都可以进入 uboot 命令行, 但是 RK3568 的 uboot 应该是被瑞芯微改过, 只能按 “CTRL+C” 组合键! 按其他按键是不能打断启动过程计入命令行的。

进入 uboot 的命令行模式以后如图 3.2.2.1 所示:

```

U-Boot 2017.09-gc2d9f80c05-220621 #alientek (May 11 2023 - 11:59:33 +0800)

Model: Rockchip RK3568 Evaluation Board
PreSerial: 2, raw, 0xfe660000
DRAM: 4 GiB
System: init
Relocation Offset: ed343000
Relocation fdt: eb9f84e0 - eb9fecd0
CR: M/C/I
Using default environment

Hotkey: ctrl+c
dwmmc@fe2b0000: 1, dwmmc@fe2c0000: 2, sdhci@fe310000: 0
Bootdev(atags): mmc 0
MMC0: HS200, 200Mhz
PartType: EFI
DM: v1
boot mode: None
FIT: no signed, no conf required
Error: find duplicate(3) dtbs
DTB: rk3568-atk-evb1-mipi-dsi-1080p#_saradc_ch2=341.dtb
HASH(c): OK
I2c0 speed: 100000Hz
PMIC: RK8090 (on=0x10, off=0x00)
vsel-gpios- not found! Error: -2
vdd_cpu init 900000 uV
vdd_logic init 900000 uV
vdd_gpu init 900000 uV
vdd_npu init 900000 uV
io-domain: OK
Could not find baseparameter partition
Model: Rockchip RK3568 ATK EVB1 DDR4 V10 Board
Rockchip UBOOT DRM driver version: v1.0.1
VOP have 2 active VP
vp0 have layer nr:3[1 3 5 ], primary plane: 5
vp1 have layer nr:3[0 2 4 ], primary plane: 4
vp2 have layer nr:0[], primary plane: 0
Using display timing dts
dsi@fe070000: detailed mode clock 75000 kHz, flags[8000000a]
    H: 1080 1128 1136 1188
    V: 1920 1936 1942 1957
bus_format: 100e
VOP update mode to: 1080x1920p0, type: MIPI1 for VP1
rockchip_vop2_init: Failed to get hdmi0_phy_pll ret=-22
rockchip_vop2_init: Failed to get hdmi1_phy_pll ret=-22
VOP VP1 enable Smart0[654x270->654x270@213x825] fmt[2] addr[0xedf2c000]
final DSI-Link bandwidth: 498 Mbps x 4
hdmi@fe0a0000 disconnected
CLK: (sync kernel. arm: enter 816000 KHz, init 816000 KHz, kernel 0N/A)
    apll 1416000 KHz
    dpll 780000 KHz
    gp11 1188000 KHz
    cp11 1000000 KHz
    np11 1200000 KHz
    vp11 600000 KHz
    hp11 24000 KHz
    pp11 200000 KHz
    armclk 1416000 KHz
    aclk_bus 150000 KHz
    pclk_bus 100000 KHz
    aclk_top_high 500000 KHz
    aclk_top_low 400000 KHz
    hclk_top 150000 KHz
    pclk_top 100000 KHz
    aclk_perimid 300000 KHz
    hclk_perimid 150000 KHz
    pclk_pmu 100000 KHz
Net: eth1: ethernet@fe010000, eth0: ethernet@fe2a0000
Hit key to stop autoboot('CTRL+C'): 0
=> <INTERRUPT>
=> <INTERRUPT>
=> <INTERRUPT>
    
```

图 3.2.2.1 uboot 启动 log 信息

从图 3.2.2.1 可以看出, 当进入到 uboot 的命令行模式以后, 左侧会出现一个 “=>”

简单介绍一下 uboot 启动过程中都打印出了哪些信息:

1、uboot 版本号, 比如图 3.2.2.1 中当前 uboot 版本号是 2017.09, 编译时间为 2023 年 5 月 11 号 11:59:33。

2、板子信息, 当前板子是瑞芯微的 RK3568 Evaluation 开发板, 这个信息是可以改的, 因为正点原子 ATK-DLRK3568 开发板是直接参考瑞芯微官方的 RK3568 开发板移植的 uboot, 所以这部分信息也就没改。

3、DDR 大小为 4GB (请参考根据个人开发板的配置, 有可能为 2G/4G 等)

4、EMMC 启动设备信息。

5、PMIC 芯片(RK809)信息。

6、DRM 信息, 也就是屏幕信息。

7、RK3568 芯片时钟信息。

8、.....

最后是“Hit key to stop autoboot(CTRL+C):”倒计时提示, 倒计时结束之前按下“CTRL+C”组合键就会进入 Uboot 命令行模式。如果在倒计时结束以后没有按下“CTRL+C”组合键, 那么 Linux 内核就会启动, Linux 内核一旦启动, uboot 就会寿终正寝。

uboot 的主要作用是引导 kernel, 我们现在已经进入 uboot 的命令行模式了, 进入命令行模式以后就可以给 uboot 发号施令了。当然了, 不能随便发号施令, 得看看 uboot 支持哪些命令, 然后使用这些 uboot 所支持的命令来做一些工作, 下一节就讲解 uboot 命令的使用。

3.3 U-Boot 命令使用

进入 uboot 的命令行模式以后输入“help”或者“?”, 然后按下回车即可查看当前 uboot 所支持的命令, 如图 3.3.1 所示:

```

=> help
?      - alias for 'help'
android_print_hdr- print android image header
atags  - Dump all atags
base   - print or set address offset
bdinfo - print Board Info structure
bidram_dump- Dump bidram layout
boot   - boot default, i.e., run 'bootcmd'
boot_android- Execute the Android Bootloader flow.
boot_fit- Boot FIT Image from memory or boot/recovery partition
bootavb - Execute the Android avb a/b boot flow.
bootd  - boot default, i.e., run 'bootcmd'
bootm  - boot application image from memory
bootp  - boot image via network using BOOTP/TFTP protocol
bootrkp - Boot Linux Image from rockchip image type
bootz  - boot Linux zImage image from memory
cmp    - memory compare
coninfo - print console devices and information
cp     - memory copy
crc32  - checksum calculation
    
```

图 3.3.1 uboot 的命令列表(部分)

图 3.3.1 中只是 uboot 的一部分命令, 具体的命令列表以实际为准。图 3.3.1 中的命令并不是 uboot 所支持的所有命令, uboot 是可配置的, 需要什么命令就使能什么命令。所以图 3.3.1 中的命令是正点原子提供的 uboot 中使能的命令, uboot 支持的命令还有很多, 而且也可以在 uboot 中自定义命令。这些命令后面都跟有命令说明, 用于描述此命令的作用, 但是命令具体怎么用呢? 我们输入“help(或?) 命令名”就可以查看命令的详细用法, 以“boot_fit”这个命令为例, 我们输入如下命令即可查看“boot_fit”这个命令的用法:

? boot_fit 或 help boot_fit

结果如图 3.3.2 所示:

```

=> ? boot_fit
boot_fit - Boot FIT Image from memory or boot/recovery partition

Usage:
boot_fit boot_fit [addr]
    
```

图 3.3.2 boot_fit 命令使用说明

图 3.3.2 列出了“boot_fit”这个命令的详细说明，其它的命令也可以使用此方法查询具体的使用方法。接下来我们学习一下一些常用的 uboot 命令。

3.3.1 查询命令

常用的和信息查询有关的命令有 3 个: bdfinfo、printenv 和 version。先来看一下 bdfinfo 命令，此命令用于查看板子信息，直接输入“bdfinfo”即可，结果如图 3.3.1.1 示：

```

=> bdfinfo
arch_number = 0x00000000
boot_params = 0x00000000
DRAM bank   = 0x00000000
-> start    = 0x00200000
-> size     = 0x08200000
DRAM bank   = 0x00000001
-> start    = 0x09400000
-> size     = 0xE6C00000
baudrate    = 1500000 bps
TLB addr    = 0xEFFF0000
relocaddr   = 0xEDD43000
reloc off   = 0xED343000
irq_sp      = 0xEB9F84D0
sp start    = 0xEB9F84D0
FB base     = 0x00000000
Early malloc usage: 36b0 / 80000
fdt_blob    = 000000000a100000
=>
    
```

图 3.3.1.1 bdfinfo 命令

从图 3.3.1.1 中可以看出 DRAM 的起始地址和大小、BOOT 参数保存起始地址、波特率、sp(堆栈指针)起始地址等信息。命令“printenv”用于输出环境变量信息，uboot 也支持 TAB 键自动补全功能，输入“print”然后按下 TAB 键就会自动补全命令。直接输入“print”也可以，因为整个 uboot 命令中只有 printenv 的前缀是“print”，所以当输入 print 以后就只有 printenv 命令了。输入“print”，然后按下回车键，环境变量如图 3.3.1.2 所示：

```

=> printenv
arch=arm
autoload=no
baudrate=1500000
board=evb_rk3568
board_name=evb_rk3568
boot_a_script=load ${devtype} ${devnum}:${distro_bootpart} ${scriptaddr} ${prefix}${script}; source ${scriptaddr}
boot_extlinux=sysboot ${devtype} ${devnum}:${distro_bootpart} any ${scriptaddr} ${prefix}extlinux/extlinux.conf
boot_net_usb_start=usb start
boot_prefixes=/ /boot/
boot_script_dhcp=boot.scr.uimg
boot_scripts=boot.scr.uimg boot.scr
boot_targets=mmc1 mmc0 mtd2 mtd1 mtd0 usb0 pxe dhcp
bootargs=storagemedia=emmc androidboot.storagemedia=emmc androidboot.mode=normal
bootcmd=boot_android ${devtype} ${devnum};boot fit;bootrkp;run distro_bootcmd;
bootcmd_dhcp=run boot_net_usb_start; if dhcp ${scriptaddr} ${boot_script_dhcp}; then source ${scriptaddr}; fi;
bootcmd_mmc0=setenv devnum 0; run mmc_boot
bootcmd_mmc1=setenv devnum 1; run mmc_boot
bootcmd_mtd0=setenv devnum 0; run mtd_boot
bootcmd_mtd1=setenv devnum 1; run mtd_boot
bootcmd_mtd2=setenv devnum 2; run mtd_boot
bootcmd_pxe=run boot_net_usb_start; dhcp; if pxe get; then pxe boot; fi
bootcmd_usb0=setenv devnum 0; run usb_boot
bootdelay=0
cpu=armv8
devnum=0
devtype=mmc
distro_bootcmd=for target in ${boot_targets}; do run bootcmd_${target}; done
    
```

图 3.3.1.2 printenv 命令部分结果

图 3.3.1.2 只是 printenv 命令的部分内容, RK3568 的 uboot 环境变量有很多。uboot 中的环境变量都是字符串, 既然叫做环境变量, 那么它的作用就和“变量”一样。比如 bootdelay 这个环境变量就表示 uboot 启动延时时间, 如果将 bootdelay 改为 5 的话就会倒计时 5s 了。uboot 中的环境变量是可以修改的, 有专门的命令来修改环境变量的值, 稍后我们会讲解。

命令 version 用于查看 uboot 的版本号, 输入“version”, uboot 版本号如图 3.3.1.3 所示:

```
=> version
U-Boot 2017.09-gc2d9f80c05-220621 #alientek (May 11 2023 - 11:59:33 +0800)

aarch64-linux-gnu-gcc (Linaro GCC 6.3-2017.05) 6.3.1 20170404
GNU ld (Linaro_Binutils-2017.05) 2.27.0.20161019
=>
```

图 3.3.1.3 version 命令结果

3.3.2 环境变量操作命令

1、修改环境变量

环境变量的操作涉及到两个命令: setenv 和 saveenv, setenv 命令用于设置或者修改环境变量的值。命令 saveenv 用于保存修改后的环境变量, 一般环境变量存放在外部 flash 中, uboot 启动的时候会将环境变量从 flash 读取到 DRAM 中。所以使用命令 setenv 修改的是 DRAM 中的环境变量值, 修改以后要使用 saveenv 命令将修改后的环境变量保存到 flash 中, 否则 uboot 下一次重启会继续使用以前的环境变量值。

瑞芯微官方 SDK 中的 uboot 默认没有使能 saveenv 命令, 但是正点原子出厂 SDK 里面已经使能了, 所以如果你用的是其他品牌的开发板, 可能无法使用 saveenv 命令。

命令 saveenv 使用起来很简单, 格式为:

```
saveenv
```

比如我们要将环境变量 bootdelay 改为 5, 就可以使用如下所示命令:

```
setenv bootdelay 5
```

```
saveenv
```

上述命令执行过程如图 3.3.2.1 所示:

```
=> setenv bootdelay 5
=> saveenv
Saving Environment to MMC...
Writing to MMC(0)... done
```

图 3.3.2.1 环境变量修改

在图 3.3.2.1 中, 当我们使用命令 saveenv 保存修改后的环境变量会有保存过程提示信息, 根据提示可以看出环境变量保存到了 MMC(0)中, 也就是 EMMC 中。修改 bootdelay 以后, 重启开发板, uboot 就是变为 5 秒倒计时, 如图 3.3.2.2 所示:

```
aclk_pdbus 500000 KHz
hclk_pdbus 198000 KHz
pclk_pdbus 99000 KHz
aclk_pdphp 297000 KHz
hclk_pdphp 198000 KHz
hclk_pdaudio 148500 KHz
hclk_pdc0re 198000 KHz
pclk_pdp0u 99000 KHz
Net: eth0: ethernet@ffc40000
Hit key to stop autoboot('CTRL+C'): 5 █
```

图 3.3.2.2 5 秒倒计时

从图 3.3.2.2 可以看出, 此时 uboot 启动倒计时变为了 5 秒。

2、新建环境变量

命令 `setenv` 也可以用于新建命令，用法和修改环境变量一样，比如我们新建一个环境变量 `author`，`author` 的值为 `'alientek'`，那么就可以使用如下命令：

```
setenv author 'alientek'
saveenv
```

`author` 命令创建完成以后重启 `uboot`，然后使用命令 `printenv` 查看当前环境变量，如图 3.3.2.3 所示：

```
=> printenv
arch=arm
author=alientek
autoload=no
baudrate=1500000
board=evb_rk3568
board_name=evb_rk3568
```

图 3.3.2.3 新建的 `author` 环境变量值

3、删除环境变量

既然可以新建环境变量，那么就可以删除环境变量，删除环境变量也是使用命令 `setenv`，要删除一个环境变量只要给这个环境变量赋空值即可，比如我们删除掉上面新建的 `author` 环境变量，命令如下：

```
setenv author
saveenv
```

上面命令中通过 `setenv` 给 `author` 赋空值，也就是什么都不写来删除环境变量 `author`。重启 `uboot` 就会发现环境变量 `author` 没有了。

3.3.3 内存操作命令

内存操作命令就是用于直接对 `DRAM` 进行读写操作的，常用的内存操作命令有 `md`、`nm`、`mm`、`mw`、`cp` 和 `cmp`，我们依次来看一下这些命令都是做什么的。

1、`md` 命令

`md` 命令用于显示内存值，格式如下：

```
md[.b, .w, .l] address [# of objects]
```

命令中的 `[.b .w .l]` 对应 `byte`、`word` 和 `long`，也就是分别以 1 个字节、2 个字节、4 个字节来显示内存值。`address` 就是要查看的内存起始地址，`[# of objects]` 表示要查看的数据长度，这个数据长度单位不是字节，而是跟你所选择的显示格式有关。比如你设置要查看的内存长度为 20(十六进制为 `0x14`)，如果显示格式为 `.b` 的话那就表示 20 个字节；如果显示格式为 `.w` 的话就表示 20 个 `word`，也就是 $20*2=40$ 个字节；如果显示格式为 `.l` 的话就表示 20 个 `long`，也就是 $20*4=80$ 个字节，另外要注意：

uboot 命令中的数字都是十六进制的！不是十进制的！

比如你想查看 `0X08300000` 开始的 20 个字节的内存值，显示格式为 `.b` 的话，应该使用如下所示命令：

```
md.b 8300000 14
```

而不是：

```
md.b 8300000 20
```

上面说了, uboot 命令里面的数字都是十六进制的, 所以可以不用写“0x”前缀, 十进制的 20 对应的十六进制为 0x14, 所以命令 md 后面的个数应该是 14, 如果写成 20 的话就表示查看 32(十六进制为 0x20)个字节的的数据。分析下面三个命令的区别:

```
md.b 8300000 10
md.w 8300000 10
md.l 8300000 10
```

上面这三个命令都是查看以 0X08300000 为起始地址的内存数据, 第一个命令以 .b 格式显示, 长度为 0x10, 也就是 16 个字节; 第二个命令以 .w 格式显示, 长度为 0x10, 也就是 16*2=32 个字节; 最后一个命令以 .l 格式显示, 长度也是 0x10, 也就是 16*4=64 个字节。这三个命令的执行结果如图 3.3.3.1 所示:

```
=> md.b 8300000 10
08300000: d0 0d fe ed 00 01 88 fe 00 00 00 38 00 01 50 c0      .....8..P.
=> md.w 8300000 10
08300000: 0dd0 edfe 0100 fe88 0000 3800 0100 c050      .....8..P.
08300010: 0000 2800 0000 1100 0000 1000 0000 0000      ..(.....
=> md.l 8300000 10
08300000: edfe0dd0 fe880100 38000000 c0500100      .....8..P.
08300010: 28000000 11000000 10000000 00000000      ..(.....
08300020: 3e380000 88500100 00000000 00000000      ..8>..P.....
08300030: 00000000 00000000 01000000 00000000      .....
```

图 3.3.3.1 md 命令使用示例

2、nm 命令

nm 命令用于修改指定地址的内存值, 命令格式如下:

```
nm [.b, .w, .l] address
```

nm 命令同样可以以 .b、.w 和 .l 来指定操作格式, 比如现在以 .l 格式修改 0X08300000 地址的数据为 0x12345678。输入命令:

```
nm.l 8300000
```

输入上述命令以后如图 3.3.3.2 所示:

```
=> nm.l 8300000
08300000: edfe0dd0 ? █
```

图 3.3.3.2 nm 命令

在图 3.3.3.2 中, 08300000 表示现在要修改的内存地址, edfe0dd0 表示地址 0X08300000 现在的数据, ‘?’ 后面就可以输入要修改后的数据 0x12345678, 输入完成以后按下回车, 然后再输入 ‘q’ 即可退出(大小写不敏感), 如图 3.3.3.3 所示:

```
=> nm.l 8300000
08300000: edfe0dd0 ? 0X12345678
08300000: 12345678 ? Q
=> █
```

图 3.3.3.3 修改内存数据

修改完成以后再使用 md 命令来查看一下有没有修改成功, 如图 3.3.3.4 所示:

```
=> md.l 8300000 1
08300000: 12345678                                xV4.
=> █
```

图 3.3.3.4 查看修改后的值

从图 3.3.3.4 可以看出, 此时地址 0X08300000 的值变为了 0X12345678。

3、mm 命令

mm 命令也是修改指定地址内存值的, 使用 mm 修改内存值的时候地址会自增, 而使用 nm

命令的话地址不会自增。比如以.l 格式修改从地址 0X08300000 开始的连续 3 个内存块(3*4=12 个字节)的数据为 0X05050505, 操作如图 3.3.3.5 所示:

```
=> mm.l 8300000
08300000: 12345678 ? 05050505
08300004: fe880100 ? 05050505
08300008: 38000000 ? 05050505
0830000c: c0500100 ? q
=>
```

图 3.3.3.5 命令 mm

从图 3.3.3.5 可以看出, 修改了地址 0X08300000、0X08300004 和 0X0830000C 的内容为 0x05050505。使用命令 md 查看修改后的值, 结果如图 3.3.3.6 所示:

```
=> md.l 8300000 3
08300000: 05050505 05050505 05050505 .....
```

图 3.3.3.6 查看修改后的内存数据

从图 3.3.3.6 可以看出内存数据修改成功。

4、mw 命令

命令 mw 用于使用一个指定的数据填充一段内存, 命令格式如下:

```
mw [.b, .w, .l] address value [count]
```

mw 命令同样以.b、.w 和.l 来指定操作格式, address 表示要填充的内存起始地址, value 为要填充的数据, count 是填充的长度。比如使用.l 格式将以 0X08300000 为起始地址的 0x10 个内存块(0x10 * 4=64 字节)填充为 0X0A0A0A0A, 命令如下:

```
mw.l 8300000 0A0A0A0A 10
```

然后使用命令 md 来查看, 如图 3.3.3.7 所示:

```
=> mw.l 8300000 0A0A0A0A 10
=> md.l 8300000 10
08300000: 0a0a0a0a 0a0a0a0a 0a0a0a0a 0a0a0a0a .....
08300010: 0a0a0a0a 0a0a0a0a 0a0a0a0a 0a0a0a0a .....
08300020: 0a0a0a0a 0a0a0a0a 0a0a0a0a 0a0a0a0a .....
08300030: 0a0a0a0a 0a0a0a0a 0a0a0a0a 0a0a0a0a .....
```

图 3.3.3.7 查看修改后的内存数据

从图 3.3.3.7 以看出内存数据修改成功。

5、cp 命令

cp 是数据拷贝命令, 用于将 DRAM 中的数据从一段内存拷贝到另一段内存中, 或者把 NorFlash 中的数据拷贝到 DRAM 中。命令格式如下:

```
cp [.b, .w, .l] source target count
```

cp 命令同样以.b、.w 和.l 来指定操作格式, source 为源地址, target 为目的地址, count 为拷贝的长度。我们使用.l 格式将 0x08300000 处的地址拷贝到 0x08300100 处, 长度为 0x10 个内存块(0x10 * 4=64 个字节), 命令如下所示:

```
cp.l 8300000 8300100 10
```

结果如图 3.3.3.8 所示:

```

=> md.l 8300000 10
08300000: 0a0a0a0a 0a0a0a0a 0a0a0a0a 0a0a0a0a .....
08300010: 0a0a0a0a 0a0a0a0a 0a0a0a0a 0a0a0a0a .....
08300020: 0a0a0a0a 0a0a0a0a 0a0a0a0a 0a0a0a0a .....
08300030: 0a0a0a0a 0a0a0a0a 0a0a0a0a 0a0a0a0a .....
=> md.l 8300100 10
08300100: 7264642c 6d69742d 00676e69 03000000 ,ddr-timing....
08300110: 04000000 3d000000 00000000 03000000 .....=.....
08300120: 04000000 4c000000 15000000 03000000 .....L.....
08300130: 04000000 5b000000 0c000000 03000000 .....[.....
=> cp.l 8300000 8300100 10
=> md.l 8300100 10
08300100: 0a0a0a0a 0a0a0a0a 0a0a0a0a 0a0a0a0a .....
08300110: 0a0a0a0a 0a0a0a0a 0a0a0a0a 0a0a0a0a .....
08300120: 0a0a0a0a 0a0a0a0a 0a0a0a0a 0a0a0a0a .....
08300130: 0a0a0a0a 0a0a0a0a 0a0a0a0a 0a0a0a0a .....
=>
    
```

图 3.3.3.8 cp 命令操作结果

在图 3.3.3.8 中，先使用 md.l 命令打印出地址 0x08300000 和 0x08300100 处的数据，然后使用命令 cp.l 将 0x08300000 处的数据拷贝到 0x08300100 处。最后使用 md.l 查看 0x08300100 处的数据有没有变化，检查拷贝是否成功。

6、cmp 命令

cmp 是比较命令，用于比较两段内存的数据是否相等，命令格式如下：

```
cmp [b, w, l] addr1 addr2 count
```

cmp 命令同样以 b、w 和 l 来指定操作格式，addr1 为第一段内存首地址，addr2 为第二段内存首地址，count 为要比较的长度。我们使用 l 格式来比较 0x08300000 和 0x08300100 这两个地址数据是否相等，比较长度为 0x10 个内存块(16 * 4=64 个字节)，命令如下所示：

```
cmp.l 8300000 8300100 10
```

结果如图 3.3.3.9 所示：

```

=> cmp.l 8300000 8300100 10
Total of 16 word(s) were the same
=>
    
```

图 3.3.3.9 cmp 命令比较结果

从图 3.3.3.9 可以看出两段内存的数据相等。我们再随便挑两段内存比较一下，比如地址 0x08300000 和 0x08300200，长度为 0x10，比较结果如图 3.3.3.10 所示：

```

=> cmp.l 8300000 8300200 10
word at 0x08300000 (0xa0a0a0a) != word at 0x08300200 (0x4000000)
Total of 0 word(s) were the same
=>
    
```

图 3.3.3.10 cmp 命令比较结果

从图 3.3.3.10 可以看出，0x08300000 处的数据和 0x08300200 处的数据就不一样。

3.3.4 网络操作命令

uboot 是支持网络的，我们在移植 uboot 的时候一般都要调通网络功能，因为在移植 linux kernel 的时候需要使用到 uboot 的网络功能做调试。uboot 支持大量的网络相关命令，比如 dhcp、ping、nfs 和 tftpboot，我们接下来依次学习一下这几个和网络有关的命令。

建议开发板和主机 PC 都连接到同一个路由器上！最后设置表 3.3.4.1 所示的几个环境变量：

环境变量	描述
ipaddr	开发板 ip 地址，可以不设置，使用 dhcp 命令来从路由器获取 IP 地址。
ethaddr	开发板的 MAC 地址，一定要设置。

gatewayip	网关地址。
netmask	子网掩码。
serverip	服务器 IP 地址，也就是 Ubuntu 主机 IP 地址，用于调试代码。

表 3.3.4.1 网络相关环境变量

表 3.3.4.1 中环境变量设置命令如下所示：**下面指令尽量手敲，复制有可能提示找不到指令！**

```
setenv ipaddr 192.168.6.38
setenv ethaddr b8:ae:1d:01:01:00 //有的 uboot 会默认设置 ethaddr，然后禁止修改，RK3568 是禁止修改的，所以这个我们可以不用设置
setenv gatewayip 192.168.6.1
setenv netmask 255.255.255.0
setenv serverip 192.168.6.227
saveenv
```

```
=> setenv ipaddr 192.168.6.38
=> setenv gatewayip 192.168.6.1
=> setenv netmask 255.255.255.0
=> setenv serverip 192.168.6.227
=>
```

注意，网络地址环境变量的设置要根据自己的实际情况，确保 Ubuntu 主机和开发板的 IP 地址在同一个网段内，比如我现在的开发板和电脑都在 192.168.6.0 这个网段内，所以设置开发板的 IP 地址为 192.168.6.38，我的 Ubuntu 主机的地址为 192.168.6.227，因此 serverip 就是 192.168.6.227。ethaddr 为网络 MAC 地址，是一个 48bit 的地址，如果在同一个网段内有多个开发板的话一定要保证每个开发板的 ethaddr 是不同的，否则通信会有问题！设置好网络相关的环境变量以后就可以使用网络相关命令了。

1、ping 命令

开发板的网络能否使用，是否可以和服务端(Ubuntu 主机)进行通信，通过 ping 命令就可以验证，直接 ping 服务器的 IP 地址即可，比如我的服务器 IP 地址为 192.168.6.227，命令如下：

```
ping 192.168.6.227
```

结果如图 3.3.4.1 所示：

```
=> ping 192.168.6.227
Using ethernet@fe010000 device
host 192.168.6.227 is alive
=>
```

图 3.3.4.1 ping 命令

从图 3.3.4.1 可以看出，192.168.6.227 这个主机存在，说明 ping 成功，uboot 的网络工作正常。

注意！只能在 uboot 中 ping 其他的机器，其他机器不能 ping uboot，因为 uboot 没有对 ping 命令做处理，如果用其他的机器 ping uboot 的话会失败！

2、dhcp 命令

dhcp 用于从路由器获取 IP 地址，前提是开发板得连接到路由器上的，如果开发板是和电脑直连的，那么 dhcp 命令就会失效。直接输入 dhcp 命令即可通过路由器获取到 IP 地址，如图 3.3.4.2 所示：

```
=> dhcp
BOOTP broadcast 1
BOOTP broadcast 2
BOOTP broadcast 3
DHCP client bound to address 192.168.6.38 (781 ms)
=>
```

图 3.3.4.2 dhcp 命令

从图 3.3.4.2 可以看出，开发板通过 dhcp 获取到的 IP 地址为 192.168.6.38。从图 3.3.4.2 可以看出，dhcp 命令不单单是获取 IP 地址，其还会通过 TFTP 来启动 linux 内核，输入“? dhcp”即可查看 dhcp 命令详细的信息，如图 3.3.4.3 所示：

```
=> ? dhcp
dhcp - boot image via network using DHCP/TFTP protocol

Usage:
dhcp [loadAddress] [[hostIPAddr:]bootfilename]
=>
```

图 3.3.4.3 dhcp 命令使用查询

3、nfs 命令

nfs(Network File System)网络文件系统，通过 nfs 可以在计算机之间通过网络来分享资源，比如我们将 linux 镜像和设备树文件放到 Ubuntu 中，然后在 uboot 中使用 nfs 命令将 Ubuntu 中的 linux 镜像和设备树下载到开发板的 DRAM 中。这样做的目的是为了更方便调试 linux 镜像和设备树，也就是网络调试，网络调试是 Linux 开发中最常用的调试方法。原因是嵌入式 linux 开发不像单片机开发，可以直接通过 JLINK 或 STLink 等仿真器将代码直接烧写到单片机内部的 flash 中，嵌入式 Linux 通常是烧写到 EMMC、NAND Flash、SPI Flash 等外置 flash 中，但是嵌入式 Linux 开发也没有 MDK, IAR 这样的 IDE，更没有烧写算法，因此不可能通过点击一个“download”按钮就将固件烧写到外部 flash 中。虽然半导体厂商一般都会提供一个烧写固件的软件，但是这个软件使用起来比较复杂，这个烧写软件一般用于量产的。其远没有 MDK、IAR 的一键下载方便，在 Linux 内核调试阶段，如果用这个烧写软件的话将会非常浪费时间，而这个时候网络调试的优势就显现出来了，可以通过网络将编译好的 linux 镜像下载到 DRAM 中，然后就可以直接运行。

我们一般使用 uboot 中的 nfs 命令将 Ubuntu 中的文件下载到开发板的 DRAM 中，在使用之前需要开启 Ubuntu 主机的 NFS 服务，并且要新建一个 NFS 使用的目录，以后所有要通过 NFS 访问的文件都需要放到这个 NFS 目录中。

接下来我们讲一下如何在 Ubuntu 中搭建 NFS 目录，要先安装并开启 Ubuntu 中的 NFS 服务，使用如下命令安装 NFS 服务：

```
sudo apt-get install nfs-kernel-server rpcbind
```

等待安装完成，安装完成以后在用户根目录下创建一个名为“linux”的文件夹，以后所有的东西都放到这个“linux”文件夹里面，在“linux”文件夹里面新建一个名为“nfs”的文件夹，如图 3.3.4.4 所示：



图 3.3.4.4 创建 linux 工作目录

图 3.3.4.4 中创建的 nfs 文件夹供 nfs 服务器使用，以后我们可以在开发板上通过网络文件系统来访问 nfs 文件夹，要先配置 nfs，使用如下命令打开 nfs 配置文件/etc/exports：

```
sudo vi /etc/exports
```

打开/etc/exports 以后在后面添加如下所示内容:

```
/home/alientek/linux/nfs *(rw, sync, no_root_squash)
```

添加完成以后的/etc/exports 如图 3.3.4.5 所示:

```
1 # /etc/exports: the access control list for filesystems which may be exported
2 #           to NFS clients.  See exports(5).
3 #
4 # Example for NFSv2 and NFSv3:
5 # /srv/homes hostname1(rw, sync, no_subtree_check) hostname2(ro, sync, no_subtree_check)
6 #
7 # Example for NFSv4:
8 # /srv/nfs4 gss/krb5i(rw, sync, fsid=0, crossmnt, no_subtree_check)
9 # /srv/nfs4/homes gss/krb5i(rw, sync, no_subtree_check)
10 #
11 /home/alientek/linux/nfs *(rw, sync, no_root_squash)
```

图 3.3.4.5 修改文件/etc/exports

Ubuntu20.04 中默认关闭了 NFS V2 版本, uboot 里面使用 nfs 命令下载可能失败, 所以需要打开 NFS V2 版本. 打开/etc/default/nfs-kernel-server, 加入下面这行:

```
RPCNFSDOPTS="--nfs-version 2,3,4 --debug --syslog"
```

如图 3.3.4.6 所示:

```
1 # Number of servers to start up
2 RPCNFSDCOUNT=8
3
4 # Runtime priority of server (see nice(1))
5 RPCNFSDPRIORITY=0
6
7 # Options for rpc.mountd.
8 # If you have a port-based firewall, you might want to set up
9 # a fixed port here using the --port option. For more information,
10 # see rpc.mountd(8) or http://wiki.debian.org/SecuringNFS
11 # To disable NFSv4 on the server, specify '--no-nfs-version 4' here
12 RPCMOUNTDOPTS="--manage-gids"
13
14 # Do you want to start the svcgssd daemon? It is only required for Kerberos
15 # exports. Valid alternatives are "yes" and "no"; the default is "no".
16 NEED_SVCGSSD=""
17
18 # Options for rpc.svcgssd.
19 RPCSVCGSSDOPTS=""
20
21 RPCNFSDOPTS="--nfs-version 2,3,4 --debug --syslog"
```

图 3.3.4.6 使能 NFS V2

最后重启 NFS 服务, 使用命令如下:

```
sudo /etc/init.d/nfs-kernel-server restart
```

至此, Ubuntu 下的 NFS 服务搭建完毕。

uboot 中的 nfs 命令格式如下所示:

```
nfs [loadAddress] [[hostIPaddr:]bootfilename]
```

loadAddress 是要保存的 DRAM 地址, [[hostIPaddr:]bootfilename]是要下载的文件地址。这里我们将正点原子官方编译出来的 Linux 内核镜像文件 boot.img 下载到开发板 DRAM 的 0X30000000 这个地址处(这个地址是笔者自己随便定义的)。正点原子编译出来的 boot.img 文件已经放到了开发板光盘中, 未替换路径为: **开发板光盘 A → 09、系统镜像 → 01、出厂系统 SDK 镜像 → boot.img**。将 boot.img 文件通过 FileZilla 发送到 Ubuntu 中的 NFS 目录下: /home/alientek/linux/nfs, 完成以后的 NFS 目录如图 3.3.4.7 所示:

```
alientek@ubuntu:~/linux/nfs$ ls
boot.img
alientek@ubuntu:~/linux/nfs$
```

图 3.3.4.7 nfs 目录

准备好以后就可以使用 nfs 命令来将 boot.img 下载到开发板 DRAM 的 0X7B3C5880 地址


```

=> md.b 30000000 100
30000000: d0 0d fe ed 00 00 06 00 00 00 00 58 00 00 04 44 .....X...D
30000010: 00 00 00 28 00 00 00 11 00 00 00 10 00 00 00 00 ...(.
30000020: 00 00 00 be 00 00 03 ec 00 00 7f 48 a3 27 20 00 .....H.'
30000030: 00 00 00 00 00 00 06 00 00 00 7f 48 a3 27 10 00 .....H.'
30000040: 00 00 00 00 00 00 06 00 00 00 00 00 00 00 00 00 .....
30000050: 00 00 00 00 00 00 00 00 00 00 01 00 00 00 00 00 .....
30000060: 00 00 00 03 00 00 00 04 00 00 00 b0 00 00 00 00 .....
30000070: 00 00 00 03 00 00 00 04 00 00 00 a6 01 67 22 00 .....g".
30000080: 00 00 00 03 00 00 00 04 00 00 00 9c 64 54 c9 d9 .....dT..
30000090: 00 00 00 03 00 00 00 1f 00 00 00 55 2d 42 6f .....U-Bo
300000a0: 6f 74 20 46 49 54 20 73 6f 75 72 63 65 20 66 69 ot FIT source fi
300000b0: 6c 65 20 66 6f 72 20 61 72 6d 00 00 00 00 00 01 le for arm.....
300000c0: 69 6d 61 67 65 73 00 00 00 00 00 01 66 64 74 00 images.....fdt.
300000d0: 00 00 00 03 00 00 00 04 00 00 00 92 00 02 2a e3 .....*.
300000e0: 00 00 00 03 00 00 00 04 00 00 00 84 00 00 08 00 .....
300000f0: 00 00 00 03 00 00 00 08 00 00 00 11 66 6c 61 74 .....flat
=>
    
```

图 3.3.4.9 下载的数据

再使用 winhex 软件来查看刚刚下载的 boot.img，检查一下前面的数据是否和图 3.3.4.9 中的一致，结果如图 3.3.4.10 所示：

boot.img																ANSI ASCII		
Offset	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F		
00000000	D0	0D	FE	ED	00	00	06	00	00	00	58	00	00	04	44		Ð pí	X D
00000010	00	00	00	28	00	00	00	11	00	00	00	10	00	00	00	00	(
00000020	00	00	00	BE	00	00	03	EC	00	00	7F	48	A3	27	20	00	% i	HÉ'
00000030	00	00	00	00	00	00	06	00	00	00	7F	48	A3	27	10	00		HÉ'
00000040	00	00	00	00	00	00	06	00	00	00	00	00	00	00	00	00		
00000050	00	00	00	00	00	00	00	00	00	00	00	01	00	00	00	00		°
00000060	00	00	00	03	00	00	00	04	00	00	00	B0	00	00	00	00		
00000070	00	00	00	03	00	00	00	04	00	00	00	A6	01	67	22	00		! g"
00000080	00	00	00	03	00	00	00	04	00	00	00	9C	64	54	C9	D9		œdTËÜ
00000090	00	00	00	03	00	00	00	1F	00	00	00	00	55	2D	42	6F		U-Bo
000000A0	6F	74	20	46	49	54	20	73	6F	75	72	63	65	20	66	69	ot	FIT source fi
000000B0	6C	65	20	66	6F	72	20	61	72	6D	00	00	00	00	00	01	le	for arm
000000C0	69	6D	61	67	65	73	00	00	00	00	00	01	66	64	74	00	images	fdt
000000D0	00	00	00	03	00	00	00	04	00	00	00	92	00	02	2A	E3		' *ã
000000E0	00	00	00	03	00	00	00	04	00	00	00	84	00	00	08	00		"
000000F0	00	00	00	03	00	00	00	08	00	00	00	11	66	6C	61	74		flat

图 3.3.4.10 winhex 查看 uImage

可以看出图 3.3.4.9 和图 3.3.4.10 中的前 0x100 个字节的数据一致，说明 nfs 命令下载到的 uImage 是正确的。

4、tftp 命令

tftp 命令的作用和 nfs 命令一样，都是用于通过网络下载东西到 DRAM 中，只是 tftp 命令使用的是 TFTP 协议，Ubuntu 主机作为 TFTP 服务器。因此需要在 Ubuntu 上搭建 TFTP 服务器，需要安装 tftp-hpa 和 tftpd-hpa，命令如下：

```

sudo apt-get install tftp-hpa tftpd-hpa
sudo apt-get install xinetd
    
```

和 NFS 一样，TFTP 也需要一个文件夹来存放文件，在用户目录下新建一个目录，命令如下：

```

mkdir /home/alientek/linux/tftpboot
chmod 777 /home/alientek/linux/tftpboot
    
```

这样我就在我的电脑上创建了一个名为 tftpboot 的目录(文件夹)，路径为/home/alientek/linux/tftpboot。注意！我们要给 tftpboot 文件夹权限，否则的话 uboot 不能从 tftpboot 文件夹里

面下载文件。

最后配置 tftp，新建文件/etc/xinetd.d/tftp，如果没有/etc/xinetd.d 目录的话自行创建，然后在里面输入如下内容：

示例代码 3.3.4.1 /etc/xinetd.d/tftp 文件内容

```

1 server tftp
2 {
3     socket_type      = dgram
4     protocol         = udp
5     wait             = yes
6     user             = root
7     server           = /usr/sbin/in.tftpd
8     server_args      = -s /home/alientek/linux/tftpboot/
9     disable          = no
10    per_source       = 11
11    cps               = 100 2
12    flags             = IPv4
13 }
```

完了以后启动 tftp 服务，命令如下：

```
sudo service tftpd-hpa start
```

打开/etc/default/tftpd-hpa 文件，将其修改为如下所示内容：

示例代码 3.3.4.2 /etc/default/tftpd-hpa 文件内容

```

1 # /etc/default/tftpd-hpa
2
3 TFTP_USERNAME="tftp"
4 TFTP_DIRECTORY="/home/alientek/linux/tftpboot"
5 TFTP_ADDRESS=":69"
6 TFTP_OPTIONS="-l -c -s"
```

TFTP_DIRECTORY 就是我们上面创建的 tftp 文件夹目录，以后我们就将所有需要通过 TFTP 传输的文件都放到这个文件夹里面，并且要给予这些文件相应的权限。

最后输入如下命令， 重启 tftp 服务器：

```
sudo service tftpd-hpa restart
```

tftp 服务器已经搭建好了，接下来就是使用了。将 boot.img 镜像文件拷贝到 tftpboot 文件夹中，并且给予 boot.img 相应的权限，命令如下：

```

cp boot.img /home/alientek/linux/tftpboot/
cd /home/alientek/linux/tftpboot/
chmod 777 boot.img
```

万事俱备，只剩验证了，uboot 中的 tftp 命令格式如下：

```
tftpboot [loadAddress] [[hostIPaddr:]bootfilename]
```

看起来和 nfs 命令格式一样的，loadAddress 是文件在 DRAM 中的存放地址，[[hostIPaddr:]bootfilename]是要从 Ubuntu 中下载的文件。但是和 nfs 命令的区别在于，tftp 命令不需要输入文件在 Ubuntu 中的完整路径，只需要输入文件名即可。比如我们现在将 tftpboot 文件夹里面的 boot.img 文件下载到开发板 DRAM 的 0X30000000 地址处，命令如下：

```
tftp 30000000 boot.img
```



```

=> ? mmc
mmc - MMC sub system

Usage:
mmc info - display info of the current MMC device
mmc read addr blk# cnt
mmc write addr blk# cnt
mmc erase blk# cnt
mmc rescan
mmc part - lists available partition on current mmc device
mmc dev [dev] [part] - show or set current mmc device [partition]
mmc list - lists available devices
mmc hwpartition [args...] - does hardware partitioning
arguments (sizes in 512-byte blocks):
[user [enh start cnt] [wrrel {on|off}]] - sets user data area attributes
[gp1|gp2|gp3|gp4 cnt [enh] [wrrel {on|off}]] - general purpose partition
[check|set|complete] - mode, complete set partitioning completed
WARNING: Partitioning is a write-once setting once it is set to complete.
Power cycling is required to initialize partitions after set to complete.
mmc testsecurestorage - test CA call static TA to store data in security
mmc testefuse - test CA call static TA, and TA read or write efuse
mmc rpmb read addr blk# cnt [address of auth-key] - block size is 256 bytes
mmc rpmb write addr blk# cnt <address of auth-key> - block size is 256 bytes
mmc rpmb key <address of auth-key> - program the RPMB authentication key.
mmc rpmb counter - read the value of the write counter
mmc setdsr <value> - set DSR register value

=>
    
```

图 3.3.5.1 mmc 命令

从图 3.3.5.1 可以看出，mmc 后面跟不同的参数可以实现不同的功能，如表 3.3.5.1 所示：

命令	描述
mmc info	输出 MMC 设备信息
mmc read	读取 MMC 中的数据。
mmc write	向 MMC 设备写入数据。
mmc rescan	扫描 MMC 设备。
mmc part	列出 MMC 设备的分区。
mmc dev	切换 MMC 设备。
mmc list	列出当前有效的所有 MMC 设备。
mmc hwpartition	设置 MMC 设备的分区。
.....

表 3.3.5.1 mmc 命令

1、mmc info 命令

mmc info 命令用于输出当前选中的 mmc info 设备的信息，输入命令“mmc info”即可，如图 3.3.5.2 所示：

```

=> mmc info
Device: sdhci@fe310000
Manufacturer ID: d6
OEM: 103
Name: A3A56
Timing Interface: HS200
Tran Speed: 200000000
Rd Block Len: 512
MMC version 5.1
High Capacity: Yes
Capacity: 57.6 GiB
Bus Width: 8-bit
Erase Group Size: 512 KiB
HC WP Group Size: 8 MiB
User Capacity: 57.6 GiB WRREL
Boot Capacity: 4 MiB ENH
RPMB Capacity: 16 MiB ENH
=>
    
```

图 3.3.5.2 mmc info 命令

从图 3.3.5.2 可以看出，当前选的 MMC 版本为 5.1，容量为 57.6GiB(EMMC 为 64GB)，速度为 200000000Hz=200MHz，8 位宽的总线。还有一个与 mmc info 命令相同功能的命令：mmcinfo，“mmc”和“info”之间没有空格。

2、mmc rescan 命令

mmc rescan 命令用于扫描当前开发板上所有的 MMC 设备，包括 EMMC 和 SD 卡，输入“mmc rescan”即可。

3、mmc list 命令

mmc list 命令用于来查看当前开发板一共有几个 MMC 设备，输入“mmc list”，结果如图 3.3.5.3 所示：

```

=> mmc list
dwmmc@fe2b0000: 1
dwmmc@fe2c0000: 2
sdhci@fe310000: 0 (eMMC)
=>
    
```

图 3.3.5.3 扫描 MMC 设备

可以看出当前开发板有三个 MMC 设备：sdhci@fec310000: 0 (eMMC)、dwmmc@fe2c0000: 2 和 dwmmc@fe2b0000:1，一共有三个个 MMC 设备，sdhci@fec310000: 0 (eMMC)是 EMMC，dwmmc@fe2b0000:1 (SD)是 SD 卡，另一个应是 SDIO WIFI 设备（若没有接 MMC 设备，这里就代表 SDIO 总线）。默认会将 EMMC 设置为当前 MMC 设备，这就是为什么输入“mmc info”查询到的是 EMMC 设备信息，而不是 SD 卡。要想查看 SD 卡信息，就要使用命令“mmc dev”来将 SD 卡设置为当前的 MMC 设备。

4、mmc dev 命令

mmc dev 命令用于切换当前 MMC 设备，命令格式如下：

```
mmc dev [dev] [part]
```

[dev]用来设置要切换的 MMC 设备号，[part]是分区号，如果不写分区号的话默认为分区 0。

使用如下命令切换到 SD 卡（请插上你的 SD 卡，如果你有的话）：

```
mmc dev 1 //切换到 SD 卡，0 为 EMMC，1 为 SD 卡
```

结果如图 3.3.5.4 所示：

```
=> mmc dev 1
switch to partitions #0, OK
mmc1 is current device
=>
```

图 3.3.5.4 切换到 SD 卡

从图 3.4.5.4 可以看出，切换到 SD 卡成功，mmc1 为当前的 MMC 设备，输入命令“mmc info”即可查看 SD 卡的信息(要插入 SD 卡)，结果如图 3.3.5.5 所示：

```
=> mmc dev 1
switch to partitions #0, OK
mmc1 is current device
=> mmc info
Device: dwmmc@fe2b0000
Manufacturer ID: 9f
OEM: 5449
Name: SD32G
Timing Interface: Legacy
Tran Speed: 52000000
Rd Block Len: 512
SD version 3.0
High Capacity: Yes
Capacity: 29.1 GiB
Bus Width: 4-bit
Erase Group Size: 512 Bytes
=>
```

图 3.3.5.5 SD 信息

从图 3.3.5.5 可以看出当前 SD 卡版本号为 3.0，容量为 29.1GiB(32GB 的 SD 卡)，4 位宽的总线。

5、mmc part 命令

有时候 SD 卡或者 EMMC 会有多个分区，可以使用命令“mmc part”来查看其分区，比如查看 EMMC 的分区情况，输入如下命令：

```
mmc dev 0 //切换到 EMMC
mmc part //查看 EMMC 分区
```

结果如图 3.3.5.6 所示：

```

=> mmc dev 0
switch to partitions #0, OK
mmc0(part 0) is current device
=> mmc part

Partition Map for MMC device 0 -- Partition Type: EFI

Part   Start LBA      End LBA      Name
Attributes
Type GUID
Partition GUID
1      0x00004000    0x00005fff   "uboot"
  attrs: 0x0000000000000000
  type:  63f95214-ef7f-4ac4-cee4-50b73f83305a
  guid:  17b18c43-1d24-4484-8883-d0c618e42411
2      0x00006000    0x00007fff   "misc"
  attrs: 0x0000000000000000
  type:  41f89008-3d1e-478d-a216-c8a14177ad67
  guid:  97c64638-8824-46d3-d83e-582a6a48d4fb
3      0x00008000    0x000027fff  "boot"
  attrs: 0x0000000000000000
  type:  548de42c-e954-4c7f-b24a-78433da0de3a
  guid:  e6edc612-f94f-44b7-b9f1-1561474fafbb
4      0x00028000    0x00047fff   "recovery"
  attrs: 0x0000000000000000
  type:  95b6230d-ae7e-43e4-df30-0b004a4882e5
  guid:  d04a9010-9d09-4d47-96ba-3fa854d3c5c5
5      0x00048000    0x00057fff   "backup"
  attrs: 0x0000000000000000
  type:  c53e440b-3208-43d3-c9b6-ee247f2711dc
  guid:  f2d9eb7f-2700-4ba8-9422-0e502db7c35c
6      0x00058000    0x00c57fff   "rootfs"
  attrs: 0x0000000000000000
  type:  1984c157-0126-4530-df80-ac1c6a725a29
  guid:  614e0000-0000-4b53-8000-1d28000054a9
7      0x00c58000    0x00c97fff   "oem"
  attrs: 0x0000000000000000
  type:  771b9f2a-581c-4528-a3e6-fe2b3cd86aa2
  guid:  7321802b-9b2c-43b7-d6d1-12af5b5de39e
8      0x00c98000    0x0733bfbf   "userdata"
  attrs: 0x0000000000000000
  type:  498d1534-a12c-492e-dd08-afb275a6c648
  guid:  d5f63744-ff57-4447-c9ee-b98e63e9b69b
=>
    
```

图 3.3.5.6 查看 EMMC 分区

从图 3.3.5.6 中可以看出，此时 EMMC 是 EFI 类型并且有 8 个分区，这个是烧写了正点原子出厂系统以后才会有的分区。

如果要图 3.3.5.6 中 EMMC 的分区 6 设置为当前 MMC 设置分区，可以使用如下命令：

```
mmc dev 0 6
```

结果如图 3.3.5.7 所示：

```

=> mmc dev 0 6
switch to partitions #6, OK
mmc0(part 6) is current device
=>
    
```

图 3.3.5.7 设置 EMMC 分区 6 为当前设备

6、mmc read 命令

mmc read 命令用于读取 mmc 设备的数据，命令格式如下：

```
mmc read addr blk# cnt
```

addr 是数据读取到 DRAM 中的地址，blk 是要读取的块起始地址(十六进制)，一个块是 512 字节，这里的块和扇区是一个意思，在 MMC 设备中我们通常说扇区，cnt 是要读取的块数量(十六进制)。比如从 EMMC 的第 1024(0x400)个块开始，读取 16(0x10)个块的数据到 DRAM 的 0XC0000000 地址处，命令如下：

```

mmc dev 0          //切换到 EMMC
mmc read 8300000 400 10 //读取数据
    
```

结果如图 3.3.5.7 所示：

```

=> mmc dev 0
switch to partitions #0, OK
mmc0(part 0) is current device
=> mmc read 8300000 400 10

MMC read: dev # 0, block # 1024, count 16 ... 16 blocks read: OK
=>
    
```

图 3.3.5.7 mmc read 命令

可以通过 md 命令来查看 0X08300000 处的数据，也就是读取到的 MMC 设备里面的数据，在这里，这些数据是随便读的，没什么意义，就不看了。

3.3.6 EXT 格式文件系统操作命令

uboot 有 ext2 和 ext4 这两种格式的文件系统的操作命令，RK3568 的系统镜像都是 ext4 格式的，所以我们重点讲解一下和 ext4 有关的三个命令：ext4ls、ext4load 和 ext4write。注意，由于只有 linux 内核、设备树和根文件系统是以 ext4 格式存放在 EMMC 中的，因此在测试 ext4 相关命令的时候要先确保 EMMC 里面烧写了完整的出厂系统！

1、ext4ls

ext4ls 命令用于查询 EXT4 格式设备的目录和文件信息，命令格式如下：

```
ext4ls <interface> [<dev[:part]>] [directory]
```

interface 是要查询的接口，比如 mmc，dev 是要查询的设备号，part 是要查询的分区，directory 是要查询的目录。比如查询 EMMC 分区 6 中的所有的目录和文件，输入命令：

```
ext4ls mmc 0:6
```

结果如图 3.3.6.1 所示：

```

=> ext4ls mmc 0:6
<DIR>      4096 .
<DIR>      4096 ..
<DIR>     16384 lost+found
<DIR>      4096 bin
             351 busybox.fragment
<SYM>         8 data
<DIR>      4096 dev
<DIR>      4096 etc
             178 init
<DIR>      4096 lib
<SYM>         3 lib64
<SYM>        11 linuxrc
<DIR>      4096 media
<SYM>        23 misc
<DIR>      4096 mnt
<DIR>      4096 oem
<DIR>      4096 opt
<DIR>      4096 proc
<DIR>      4096 rockchip_test
<DIR>      4096 root
<DIR>      4096 run
<DIR>      4096 sbin
<SYM>        10 sdcard
<DIR>      4096 sys
<DIR>      4096 system
<DIR>      4096 tmp
<SYM>         9 udisk
<DIR>      4096 userdata
<DIR>      4096 usr
<DIR>      4096 var
<SYM>         6 vendor
<DIR>      4096 .cache
             3366 gst.sh
<DIR>      4096 .config
             43106 .bluetooth.log
             441044 .record.wav
=>
    
```

图 3.3.6.1 EMMC 分区 2 文件查询

从上图可以看出, emmc 的分区 6 存放的就是我们的根文件系统。

2、ext4load 命令

extload 命令用于将指定的文件读取到 DRAM 中, 命令格式如下:

```
fatload <interface> [<dev[:part]>] [<addr>] [<filename>] [bytes [pos]]]
```

interface 为接口, 比如 mmc, dev 是设备号, part 是分区, addr 是保存在 DRAM 中的起始地址, filename 是要读取的文件名字。bytes 表示读取多少字节的数据, 如果 bytes 为 0 或者省略的话表示读取整个文件。pos 是要读的文件相对于文件首地址的偏移, 如果为 0 或者省略的话表示从文件首地址开始读取。我们将图 3.3.6.1 中 EMMC 分区 6 中的 linuxrc 文件读取到 DRAM 中的 0X08300000 地址处, 命令如下:

```
ext4load mmc 0:6 8300000 linuxrc
```

操作过程如图 3.3.6.2 所示:

```
=> ext4load mmc 0:6 8300000 linuxrc
754880 bytes read in 10 ms (72 MiB/s)
=>
```

图 3.3.6.2 读取过程

从上图可以看出在 10ms 内读取了 754880 个字节的数据, 速度为 72MiB/s, 速度是非常快的, 因为这是从 EMMC 里面读取的, 而 EMMC 是 8 位的, 速度肯定会很快。

3.3.7 BOOT 操作命令

uboot 的本质工作是引导 Linux, 所以 uboot 肯定有相关的 boot(引导)命令来启动 Linux。常用的跟 boot 有关的命令有: boot_fit 和 boot。

1、boot_fit 命令

大家如果学过 I.MX6U 或者 STM32MP1 的话, 应该知道 uboot 使用 bootm 或者 bootz 这两个命令启动内核, 需要提供 Linux 编译出来的 zImage 或 uImage 以及设备树文件, 然后使用 bootm 或 bootz 启动。但是 RK3568 最终的系统烧写文件只有一个 boot.img, Image 和设备树文件全部打包进 boot.img 这一个文件里面, 所以就不能用 bootm 或 bootz, 要用到 boot_fit 命令。

boot_fit 命令格式如下:

```
boot_fit [addr]
```

addr 是可选的, 也就是 boot.img 在 DRAM 中的位置。可以不需要, boot_fit 默认会从相应的 boot 分区里面读取 boot.img 然后解析启动 Linux 内核, 比如当我们把系统烧写到 EMMC 里面以后, 只需要一个 boot_fit 命令就可以完成 Linux 内核和设备树的提取、启动。

如果想要从网络启动系统, 那么先把 boot.img 放到 Ubuntu 的 tftpboot 文件夹中, 然后用 tftp 命令将 boot.img 下载到开发板的 DRAM 中, 然后使用 boot_fit 命令启动。首先是要先将 boot.img 通过 tftp 下载到合适的 DRAM 地址处, 这里需要用到 sysmem_search 命令找到一个合适的存储起始地址, sysmem_search 命令格式如下:

```
sysmem_search size
```

size 就是要获取的内存大小, 为十六进制的。一般来说就是 boot.img 大小, 但是每次重新编译 Linux 内核以后 boot.img 大小都会变, 每次都要获取内存起始地址太麻烦了。我们就直接获取一个 25MB 的空间就行了, 所以 25MB=0X1900000, 输入如下命令:

```
sysmem_serach 1900000
```

结果如图 3.3.7.1 所示:

```
=> system_search 1900000
Systemem: Available region at address: 0xe9ef6400
=>
```

图 3.3.7.1 得到的内存起始地址

从图 3.3.7.1 可以看出, 我当前得到的地址为 0XE9EF6400, 大家以自己实际得到的地址为准! 接下来就是通过 tftp 将 boot.img 下载到 0XE9EF6400 地址处, 然后通过 boot_fit 命令启动, 整个的命令如下:

```
tftp E9EF6400 boot.img
boot_fit E9EF6400
```

命令运行结果如图 3.3.7.2 所示:

导”和“命令”，说明这个环境变量保存着引导命令，其实就是多条启动命令的集合，具体的引导命令内容是可以修改的。

RK3568 开发板 bootcmd 默认值如图 3.3.7.3 所示：

```

=> print bootcmd
bootcmd=boot_fit;boot_android ${devtype} ${devnum};
=>

=> print bootcmd
bootcmd=boot_android ${devtype} ${devnum};boot_fit;bootrkp;run distro_bootcmd;
=>
    
```

图 3.3.7.3 bootcmd 默认值

可以看出 bootcmd 默认值为“boot_fit;boot_android \${devtype} \${devnum};bootrkp;run distro_bootcmd”，其中 devtype 为 mmc，devnum 为 0，所以简化一下就是“boot_fit;boot_android mmc 0;”。也就是有两种启动内核的方式：boot_fit 和 boot_android，其中 boot_android 在 RK3568 里面无效，因此实际有效的只有一个 boot_fit。后面的 bootrkp;run distro_bootcmd 未定义，所以无效。

因此在 RK3568 中，bootcmd 就是直接调用 boot_fit 命令来启动 Linux 系统的。uboot 倒计时结束以后会默认运行 bootcmd 环境变量里面的命令，所以 boot_fit 也就会默认执行。

3.3.8 其他常用命令

uboot 中还有其他一些常用的命令，比如 reset、go、run 和 mtest 等。

1、reset 命令

reset 命令顾名思义就是复位的，输入“reset”即可复位重启，如图 3.3.8.1 所示：

```

=> reset
DDR Version V1.13 20220218
In
ddrconfig:0
LPDDR4X, 324MHz
BW=32 Col=10 Bk=8 CS0 Row=17 CS=1 Die BW=16 Size=4096MB
tdqss: cs0 dqs0: 48ps, dqs1: -72ps, dqs2: -72ps, dqs3: -144ps,

change to: 324MHz
PHY drv:clk:36,ca:36,DQ:29,odt:0
vrefinner:24%, vrefout:41%
dram drv:40,odt:0
clk skew:0x61

change to: 528MHz
PHY drv:clk:36,ca:36,DQ:29,odt:0
vrefinner:24%, vrefout:41%
dram drv:40,odt:0
clk skew:0x58

change to: 780MHz
PHY drv:clk:36,ca:36,DQ:29,odt:0
vrefinner:24%, vrefout:41%
dram drv:40,odt:0
clk skew:0x58

change to: 1560MHz(final freq)
PHY drv:clk:36,ca:36,DQ:29,odt:60
vrefinner:16%, vrefout:22%
dram drv:40,odt:80
vref_ca:00000071
clk skew:0x26
cs 0:
the read training result:
DQS0:0x32, DQS1:0x32, DQS2:0x36, DQS3:0x38,
    
```

图 3.3.8.1 reset 命令运行结果

2、go 命令

go 命令用于跳到指定的地址处执行应用，命令格式如下：

go addr [arg ...]

addr 是应用在 DRAM 中的首地址。

3、run 命令

run 命令用于运行环境变量中定义的命令，比如可以通过“run bootcmd”来运行 bootcmd 中的启动命令，但是 run 命令最大的作用在于运行我们自定义的环境变量。

至此，uboot 常用的命令就讲解完了，如果要使用 uboot 的其他命令，可以查看 uboot 中的帮助信息，或者上网查询一下相应的资料。

3.3.9 MII 命令使用说明

MI I 命令是网络相关命令，主要用于读取网络 PHY 芯片寄存器，在 uboot 中调试网络 PHY 芯片的时候非常有用！MI I 是一系列命令，如图 3.3.9.1 所示：

```

=> mii
mii - MII utility commands

Usage:
mii device                - list available devices
mii device <devname>     - set current device
mii info <addr>          - display MII PHY info
mii read <addr> <reg>    - read MII PHY <addr> register <reg>
mii write <addr> <reg> <data> - write MII PHY <addr> register <reg>
mii modify <addr> <reg> <data> <mask> - modify MII PHY <addr> register <reg>
                                     updating bits identified in <mask>
mii dump <addr> <reg>    - pretty-print <addr> <reg> (0-5 only)
Addr and/or reg may be ranges, e.g. 2-7.
=>
    
```

图 3.3.9.1 mii 系列命令

我们使用“mii info”命令，“mii info”命令格式如下：

```
mii info <addr>
```

其中 addr 就是 PHY 芯片地址，RK3568 开发板上网络 PHY 地址为 0X01，输入如下命令：

```
mii info 0x1
```

结果如图 3.3.9.2 所示：

```

=> mii info 1
PHY 0x01: OUI = 0x13D47A, Model = 0x11, Rev = 0x0B, 100baseT, FDX
=>
    
```

图 3.3.9.2 mii info 命令

也可以使用“mii dump”直接打印出 PHY 的 0~5 寄存器值，输入如下命令：

```
mii dump 1 0-5
```

其中 1 表示 PHY 地址为 1，0-5 表示读取 0-5 这 6 个寄存器，结果如图 3.3.9.3 所示：

```

=> mii dump 1 0-5
0.      (1140)
      (8000:0000) 0.15 = 0 -- PHY control register --
      (4000:0000) 0.14 = 0 reset
      (2040:0040) 0. 6,13 = b10 loopback
      (1000:1000) 0.12 = 1 speed selection = 1000 Mbps
      (0800:0000) 0.11 = 0 A/N enable
      (0400:0000) 0.10 = 0 power-down
      (0200:0000) 0. 9 = 0 isolate
      (0100:0100) 0. 8 = 0 restart A/N
      (0080:0000) 0. 7 = 1 duplex = full
      (003f:0000) 0. 5- 0 = 0 collision test enable
      (reserved)
1.      (796d)
      (8000:0000) 1.15 = 0 -- PHY status register --
      (4000:4000) 1.14 = 0 100BASE-T4 able
      (2000:2000) 1.13 = 1 100BASE-X full duplex able
      (1000:1000) 1.12 = 1 100BASE-X half duplex able
      (0800:0800) 1.11 = 1 10 Mbps full duplex able
      (0400:0000) 1.10 = 1 10 Mbps half duplex able
      (0200:0000) 1. 9 = 0 100BASE-T2 full duplex able
      (0100:0100) 1. 8 = 0 100BASE-T2 half duplex able
      (0080:0000) 1. 7 = 1 extended status
      (0040:0040) 1. 6 = 0 (reserved)
      (0020:0020) 1. 5 = 1 MF preamble suppression
      (0010:0000) 1. 4 = 0 A/N complete
      (0008:0008) 1. 3 = 0 remote fault
      (0004:0004) 1. 2 = 1 A/N able
      (0002:0000) 1. 1 = 1 link status
      (0001:0001) 1. 0 = 0 jabber detect
      (reserved)
2.      (4f51)
      (ffff:4f51) 2.15- 0 = 20305 -- PHY ID 1 register --
      OUI portion
3.      (e91b)
      (fc00:e800) 3.15-10 = 58 -- PHY ID 2 register --
      OUI portion
      (03f0:0110) 3. 9- 4 = 17 manufacturer part number
      (000f:000b) 3. 3- 0 = 11 manufacturer rev. number
4.      (11e1)
      (8000:0000) 4.15 = 0 -- Autonegotiation advertisement register --
      (4000:0000) 4.14 = 0 next page able
      (2000:0000) 4.13 = 0 (reserved)
      (1000:1000) 4.12 = 0 remote fault
      (0800:0000) 4.11 = 1 (reserved)
      (0400:0000) 4.10 = 0 asymmetric pause
      (0200:0000) 4. 9 = 0 pause enable
      (0100:0100) 4. 8 = 0 100BASE-T4 able
      (0080:0080) 4. 7 = 1 100BASE-TX full duplex able
      (0040:0040) 4. 6 = 1 100BASE-TX able
      (0020:0020) 4. 5 = 1 10BASE-T full duplex able
      (001f:0001) 4. 4- 0 = 1 10BASE-T able
      selector = IEEE 802.3
5.      (c5e1)
      (8000:8000) 5.15 = 0 -- Autonegotiation partner abilities register --
      (4000:4000) 5.14 = 1 next page able
      (2000:0000) 5.13 = 1 acknowledge
      (1000:0000) 5.12 = 0 remote fault
      (0800:0000) 5.11 = 0 (reserved)
      (0400:0400) 5.10 = 0 asymmetric pause able
      (0200:0000) 5. 9 = 1 pause able
      (0100:0100) 5. 8 = 0 100BASE-T4 able
      (0080:0080) 5. 7 = 1 100BASE-X full duplex able
      (0040:0040) 5. 6 = 1 100BASE-TX able
      (0020:0020) 5. 5 = 1 10BASE-T full duplex able
      (001f:0001) 5. 4- 0 = 1 10BASE-T able
      selector = IEEE 802.3
=>
    
```

大家也可以使用“mii read”命令读取 PHY 芯片的其他寄存器。或者使用“mii write”命令向指定的寄存器写入一个数据。

第四章 Linux 驱动开发准备工作

在正式开始学习 Linux 驱动开发之前,有一些准备工作要先处理好,比如交叉编译器安装, Linux 内核与设备树的确定, ADB 使用等。

4.1 Linux 内核编译

4.1.1 编译 Linux 内核

本书 Linux 驱动开发全部基于正点原子出厂系统，包括根文件系统。但是在 Linux 驱动开发过程中肯定要涉及到修改或编译 Linux 内核源码以及设备树。关于正点原子出厂 SDK 包中的 Linux 内核编译方法请参考[开发板光盘→09、用户手册 →【正点原子】ATK-DLRK3568 嵌入式 linux 系统开发手册.pdf](#)中的 6.2 小节。

我们最终烧写到开发板里面的是编译出来的 boot.img 文件，boot.img 就是将编译出来的 Image 和 rk3568-atk-evb1-ddr4-v10-linux.dtb 打包在一起，rk3568-atk-evb1-ddr4-v10-linux.dtb 就是 arch/arm/boot/dts/ rk3568-atk-evb1-ddr4-v10-linux.dts 设备树文件编译出来的。所以我们后面如果要修改设备树，那么修改的就是 rk3568-atk-evb1-ddr4-v10-linux.dts 这个文件。请根据自己的 MIPI 屏幕分辨率大小，修改 rk3568-screen_choose.dtsi 这个设备树文件，开启对应的宏。

4.1.2 关闭内核 log 时间戳

正点原子出厂系统内核运行的时候默认会打印出时间戳，如图 4.1.2.1 所示：

```
[ 1.407688] hub 1-1:1.0: USB hub found
[ 1.407954] hub 1-1:1.0: 4 ports detected
[ 1.421346] EXT4-fs (mmcblk0p6): recovery complete
[ 1.423805] EXT4-fs (mmcblk0p6): mounted filesystem with ordered data mode. Opts: (null)
[ 1.423896] VFS: Mounted root (ext4 filesystem) readonly on device 179:6.
[ 1.424638] devtmpfs: mounted
[ 1.426187] Freeing unused kernel memory: 1024K
[ 1.426676] Run /sbin/init as init process
[ 1.541051] EXT4-fs (mmcblk0p6): re-mounted. Opts: (null)
[ 1.790994] phy phy-ff4c0000.usb2-phy.1: charger = USB_CDP_CHARGER
```

图 4.1.2.1 内核 log 时间戳

图 4.1.2.1 中最前面的就是 log 信息的时间戳，在调试的时候这个时间信息看起来比较“烦人”，如需要需要关闭，我们可以通过配置内核将其关闭，配置路径如下：

进入 kernel，执行下面的命令配置内核。

```
make ARCH=arm64 rockchip_linux_defconfig
```

打开图形菜单，执行下面的指令。

```
make ARCH=arm64 menuconfig
```

按如下步骤取消选中内核打印。

```
-> Kernel hacking
```

```
    -> printk and dmesg options
```

```
        ->Show timing information on printkts //取消选中
```

如图 4.1.2.2 所示：

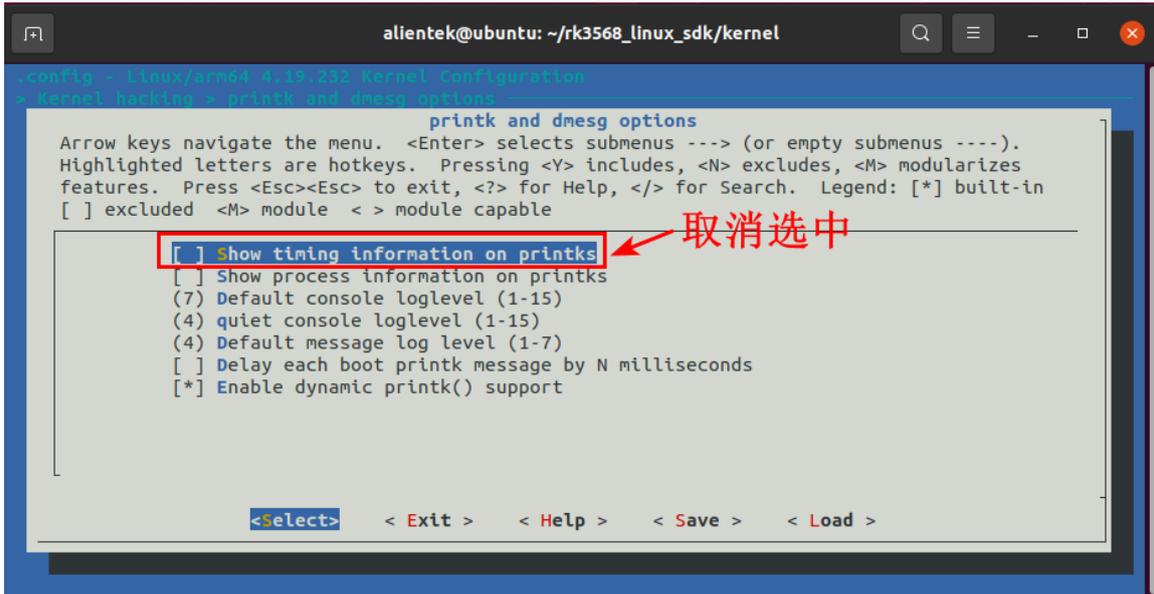


图 4.1.2.2 取消 printk 时间戳

重新编译并烧写内核即可。

4.2 根文件系统确认

4.2.1 创建/lib/modules/4.19.232 目录

我们需要在根文件系统中创建/lib/modules/4.19.232 目录，因为后面驱动开发都是将驱动编译成模块，然后放到/lib/modules/4.19.232 目录中。其中 4.19.232 是所使用的 Linux 内核版本号，RK3568 所使用的 linux 内核版本号为 4.19.232，所以就是/lib/modules/4.19.232 目录。

出厂系统有/lib/moduels 这个目录，大家只需要在/lib/moduels 里面创建一个名为“4.19.232”子目录就行了，完成以后如图 4.2.1.1 所示：

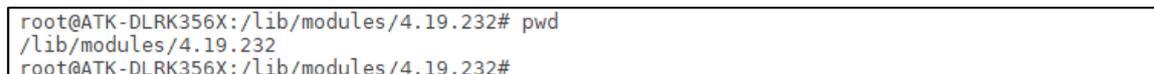


图 4.2.1.1 /lib/modules/4.19.232 目录

4.2.2 检查相关命令是否存在

我们需要使用 depmod、modprobe 或 insmod 这三个命令来加载驱动模块，所以根文件系统要存在这两个命令。卸载驱动模块的时候需要用到 rmmod 命令，所以需要确保 depmod、modprobe、insmod 和 rmmod 这四个命令都存在。

4.3 Ubuntu 下使用 ADB 向开发板发送文件

我们一般都是在 Ubuntu 下进行驱动和应用的编写与编译，最终肯定要将编译出来的应用文件或驱动文件发送到开发板中运行测试。

关于开发板的 adb 安装与使用请参考[开发板光盘→10、用户手册→【正点原子】adb 工具使用说明.pdf](#)。

在驱动开发中，我们需要使用 adb 将开发板连接到 Ubuntu 下，然后使用如下“adb push”命令将文件通过 adb 发送到开发板。

我们在 Ubuntu 下随便新建一个名为“adbttest.txt”的空文件，然后将其通过 adb 发送到开发

板的/lib/modules/4.19.232 目录下。

1、开发板通过 ADB 连接到 Ubuntu

首先使用 adb 命令将开发板连接到 Ubuntu，先用“adb devices”命令查看一下能不能找到开发板，如果能找到的话就会例举出当前 Ubuntu 下所有连接的 adb 设备，如图 4.3.1 所示：

```
alientek@ubuntu:~/rk3568_linux_sdk/kernel$ adb devices
List of devices attached
831506727c0ec4d5      device
alientek@ubuntu:~/rk3568_linux_sdk/kernel$
```

图 4.3.1 当前 adb 设备

图 4.3.1 中的 831506727c0ec4d5 就是我的开发板，如果找不到对应的设备，请检查 Ubuntu 下相关 adb 命令有没有安装成功，或者 USB OTG 线有没有连接好。

使用“adb push”将前面创建的 adbttest.txt 文件发动到开发板中，输入如下命令：

```
adb push adbttest.txt /lib/modules/4.19.232/
```

过程如图 4.3.2 所示：

```
alientek@ubuntu:~/rk3568_linux_sdk/kernel$ adb push adbttest.txt /lib/modules/4.19.232/
adbttest.txt: 1 file pushed.
alientek@ubuntu:~/rk3568_linux_sdk/kernel$
```

图 4.3.2 adb push 运行过程

从图 4.3.2 中可以看出，adbttest.txt 文件已经 push 成功。到开发板中查看一下 /lib/modules/4.19.232 目录下是否有 adbttest.txt 这个文件，如图 4.3.3 所示：

```
root@ATK-DLRK356X:/lib/modules/4.19.232# ls
adbttest.txt
root@ATK-DLRK356X:/lib/modules/4.19.232#
```

图 4.3.3 发送到开发板里面的 adbttest.txt 文件

4.4 安装驱动开发所使用的交叉编译器

编译驱动肯定要用到交叉编译器，虽然 SDK 包里面已经提供了交叉编译器，但是需要先编译整个 SDK 包，而且里面的交叉编译期用起来也不方便，所以正点原子专门定制了一套交叉编译器，大家直接安装就可以很方便的使用。交叉编译器安装包存放路径：**开发板光盘→05、开发工具→交叉编译工具→atk-dlrk3568-toolchain-arm-buildroot-linux-gnueabi-hf-x86_64.run**。如图 4.4.1 所示：

名称	修改日期	类型	大小
atk-dlrk3568-toolchain-arm-buildroot-linux-gnueabi-hf-x86_64.run	2023/3/31 10:58	RUN 文件	221,549 KB
使用说明.txt	2023/5/24 15:15	文本文档	1 KB

图 4.4.1 交叉编译工具链

如果你已经在 Ubuntu 中安装了上面的交叉编译工具链那么就不需要再安装了，将 atk-dlrk3568-toolchain-arm-buildroot-linux-gnueabi-hf-x86_64.run 拷贝到 Ubuntu 中。然后使用如下命令给予可执行权限：

```
chmod a+x atk-dlrk3568-toolchain-arm-buildroot-linux-gnueabi-hf-x86_64.run
```

如图 4.4.2 所示：

```
alientek@ubuntu:~/tools$ chmod a+x atk-dlrk3568-toolchain-arm-buildroot-linux-gnueabi-hf-x86_64.run
alientek@ubuntu:~/tools$ ls
atk-dlrk3568-toolchain-arm-buildroot-linux-gnueabi-hf-x86_64.run
alientek@ubuntu:~/tools$
```

图 4.4.2 给予可执行权限

输入如下命令，

```
./atk-dlrk3568-toolchain-arm-buildroot-linux-gnueabi-hf-x86_64.run
```

当出现“Enter target directory for toolchain (default: /opt/atk-dlrk3568-toolchain):”时，表示让你确认安装路径，默认安装路径是/opt/atk-dlrk356x-toolchain，不要修改这个路径，直接按下回车键使用这个默认路径即可。如图 4.4.3 所示：

```
alientek@ubuntu:~/tools$ ./atk-dlrk3568-toolchain-arm-buildroot-linux-gnueabi-hf-x86_64.run
ATK-DLRK356X toolchain installer version 1.0.0 Generated by Buildroot!
=====
Enter target directory for toolchain (default: /opt/atk-dlrk356x-toolchain):
```

图 4.4.3 确认默认安装路径

在图 4.4.3 中输入回车键，进入图 4.4.4 所示步骤：

```
alientek@ubuntu:~/tools$ ./atk-dlrk3568-toolchain-arm-buildroot-linux-gnueabi-hf-x86_64.run
ATK-DLRK356X toolchain installer version 1.0.0 Generated by Buildroot!
=====
Enter target directory for toolchain (default: /opt/atk-dlrk356x-toolchain):
You are about to install the toolchain to "/opt/atk-dlrk356x-toolchain". Proceed[Y/n]? y
```

图 4.4.4 确认安装目录

图 4.4.4 是让你确认安装，输入‘Y’即可，接下来会让你输入密码，自行输入密码即可，如图 4.4.5 所示：

```
alientek@ubuntu:~/tools$ ./atk-dlrk3568-toolchain-arm-buildroot-linux-gnueabi-hf-x86_64.run
ATK-DLRK356X toolchain installer version 1.0.0 Generated by Buildroot!
=====
Enter target directory for toolchain (default: /opt/atk-dlrk356x-toolchain):
You are about to install the toolchain to "/opt/atk-dlrk356x-toolchain". Proceed[Y/n]? y
[sudo] password for alientek:
```

图 4.4.5 输入密码

最后等待安装完成，安装完成以后如图 4.4.6 所示：

```
alientek@ubuntu:~/tools$ ./atk-dlrk3568-toolchain-arm-buildroot-linux-gnueabi-hf-x86_64.run
ATK-DLRK356X toolchain installer version 1.0.0 Generated by Buildroot!
=====
Enter target directory for toolchain (default: /opt/atk-dlrk356x-toolchain):
You are about to install the toolchain to "/opt/atk-dlrk356x-toolchain". Proceed[Y/n]? y
[sudo] password for alientek:
Extracting toolchain.....done
Relocating the toolchain to /opt/atk-dlrk356x-toolchain...
Toolchain has been successfully set up and is ready to be used.
Each time you wish to use the Toolchain in a new shell session, you need to source the environment setup script e.g.
$ . export PATH=$PATH:/opt/atk-dlrk356x-toolchain/usr/bin
alientek@ubuntu:~/tools$
```

图 4.4.6 安装完成

安装完成以后可以进入到/opt/atk-dlrk356x-toolchain/bin/目录下，看一下安装好的交叉编译器，如图 4.4.7 所示：

```
alientek@ubuntu:~/tools$ ls /opt/atk-dlrk356x-toolchain/bin/
2to3                                autopoint                            linux64                             python3.8
2to3-3.8                            autoreconf                           localedef                           python3.8-config
aarch64-buildroot-linux-gnu-addr2line  autoscan                            logger                               python3-config
aarch64-buildroot-linux-gnu-ar        autoupdate                           look                                 python-config
aarch64-buildroot-linux-gnu-as        bison                                lowntfs-3g                          python-freeze-importlib
aarch64-buildroot-linux-gnu-c++       cal                                   lsattr                               qdbuscpp2xml
aarch64-buildroot-linux-gnu-c++_br_real  captoinfo                            lsblk                                qdbusxml2cpp
aarch64-buildroot-linux-gnu-cc        chatr                                 lscpu                                qgltf
aarch64-buildroot-linux-gnu-cc_br_real  choom                                 lstpc                                qlalr
aarch64-buildroot-linux-gnu-c++filt    cjpeg                                 lsirq                                 qmake
aarch64-buildroot-linux-gnu-cpp       clear                                  lslocks                              qmlcacheagen
aarch64-buildroot-linux-gnu-cpp_br_real  col                                   lsmem                               qmlformat
```

图 4.4.7 安装好的交叉编译器

注意，这里大家就不能直接通过输入“aarch64-buildroot-linux-gnu-gcc -v”这样来使用交叉编译器了，而是需要输入绝对路径，也就是：

```
/opt/atk-dlrk356x-toolchain/bin/aarch64-buildroot-linux-gnu-gcc -v
```

结果如图 4.4.8 所示：

```
allentek@ubuntu:~/tools$ /opt/atk-dlrk356x-toolchain/bin/aarch64-buildroot-linux-gnu-gcc -v
使用内建 specs。
COLLECT_GCC=/opt/atk-dlrk356x-toolchain/bin/aarch64-buildroot-linux-gnu-gcc.br_real
COLLECT_LTO_WRAPPER=/opt/atk-dlrk356x-toolchain/bin/../libexec/gcc/aarch64-buildroot-linux-gnu/10.3.0/lto-wrapper
目标: aarch64-buildroot-linux-gnu
配置为: ./configure --prefix=/home/allentek/ATK-DLRK3568/buildroot/output/rockchip_rk3568/host --sysconfdir=/home/allentek/ATK-DLRK3568/buildroot/output/rockchip_rk3568/host/etc --enable-static --target=aarch64-buildroot-linux-gnu --with-sysroot=/home/allentek/ATK-DLRK3568/buildroot/output/rockchip_rk3568/host/aarch64-buildroot-linux-gnu/sysroot --enable-__cxa_atexit --with-gnu-ld --disable-libssp --disable-multilib --disable-decimal-float --with-gmp=/home/allentek/ATK-DLRK3568/buildroot/output/rockchip_rk3568/host --with-mpc=/home/allentek/ATK-DLRK3568/buildroot/output/rockchip_rk3568/host --with-mpfr=/home/allentek/ATK-DLRK3568/buildroot/output/rockchip_rk3568/host --with-pkgversion='Buildroot 2018.02-rc3-ge067d8b0-dirty' --with-bugurl=http://bugs.buildroot.net/ --without-zstd --disable-libquadmath --disable-libquadmath-support --enable-tls --enable-threads --without-isl --without-cloog --with-abi=lp64 --with-cpu=cortex-a55 --enable-languages=c,c++ --with-build-time-tools=/home/allentek/ATK-DLRK3568/buildroot/output/rockchip_rk3568/host/aarch64-buildroot-linux-gnu/bin --enable-shared --disable-libgomp
线程模型: posix
Supported LTO compression algorithms: zlib
gcc 版本 10.3.0 (Buildroot 2018.02-rc3-ge067d8b0-dirty)
allentek@ubuntu:~/tools$
```

图 4.4.8 交叉编译器工具链

当然了你也可以自行把/opt/atk-dlrk356x-toolchain/bin/这个路径添加到环境变量，这样在使用交叉编译期的时候就不需要指定绝对路径了，但是不建议新手这么做，因为见过太多新手修改环境变量将系统改崩溃的！

若要卸载交叉编译器，进入安装目录/opt 下直接删除即可，命令如下：

```
sudo rm -f atk-dlrk356x-toolchain/
```

本教程后面将全部采用绝对路径访问的方式来使用交叉编译器！

第五章 字符设备驱动开发

本章我们从 Linux 驱动开发中最基础的字符设备驱动开始，重点学习 Linux 下字符设备驱动开发框架。本章会以一个虚拟的设备为例，讲解如何进行字符设备驱动开发，以及如何编写测试 APP 来测试驱动工作是否正常，为以后的学习打下坚实的基础。

5.1 字符设备驱动简介

字符设备是 Linux 驱动中最基本的一类设备驱动，字符设备就是一个一个字节，按照字节流进行读写操作的设备，读写数据是分先后顺序的。比如我们最常见的点灯、按键、IIC、SPI，LCD 等等都是字符设备，这些设备的驱动就叫做字符设备驱动。

在详细的学习字符设备驱动架构之前，我们先来简单的了解一下 Linux 下的应用程序是如何调用驱动程序的，Linux 应用程序对驱动程序的调用如图 5.1.1 所示：

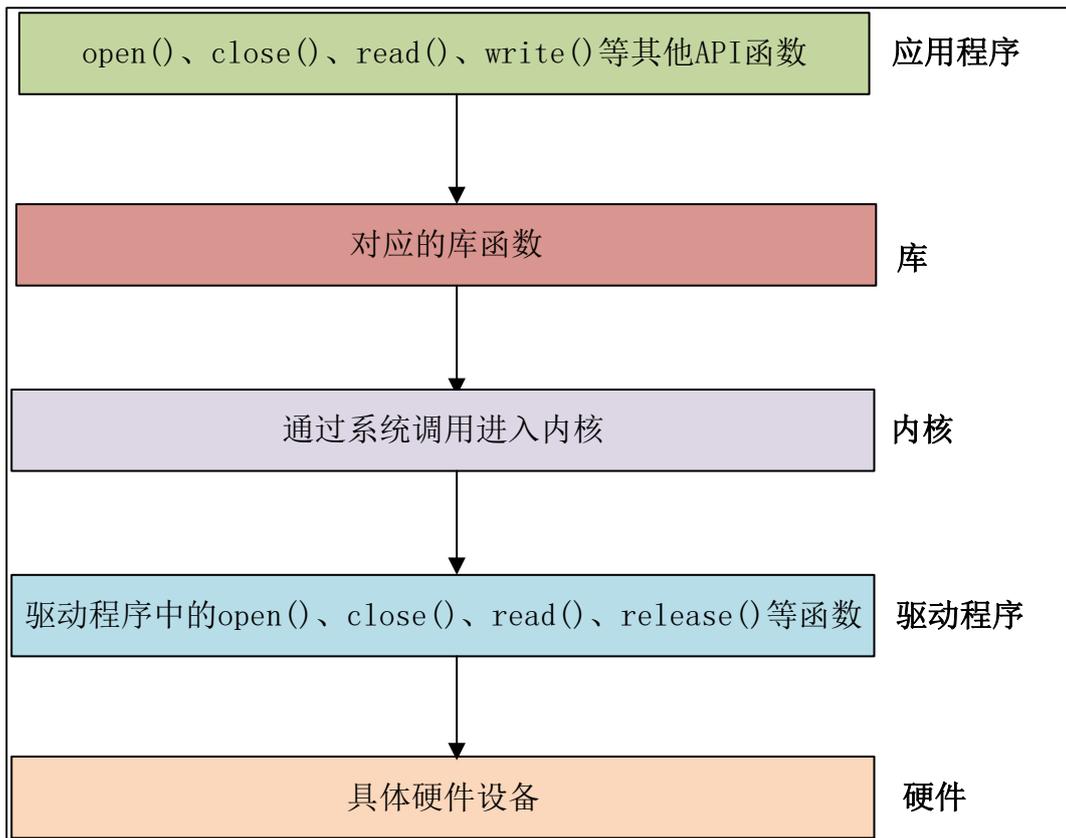


图 5.1.1 Linux 应用程序对驱动程序的调用流程

在 Linux 中一切皆为文件，驱动加载成功以后会在“/dev”目录下生成一个相应的文件，应用程序通过对这个名为“/dev/xxx” (xxx 是具体的驱动文件名字)的文件进行相应的操作即可实现对硬件的操作。比如现在有个叫做/dev/led 的驱动文件，此文件是 led 灯的驱动文件。应用程序使用 open 函数来打开文件/dev/led，使用完成以后使用 close 函数关闭/dev/led 这个文件。open 和 close 就是打开和关闭 led 驱动的函数，如果要点亮或关闭 led，那么就使用 write 函数来操作，也就是向此驱动写入数据，这个数据就是要关闭还是要打开 led 的控制参数。如果要获取 led 灯的状态，就用 read 函数从驱动中读取相应的状态。

应用程序运行在用户空间，而 Linux 驱动属于内核的一部分，因此驱动运行于内核空间。当我们在用户空间想要实现对内核的操作，比如使用 open 函数打开/dev/led 这个驱动，因为用户空间不能直接对内核进行操作，因此必须使用一个叫做“系统调用”的方法来实现从用户空间“陷入”到内核空间，这样才能实现对底层驱动的操作。open、close、write 和 read 等这些函数是由 C 库提供的，在 Linux 系统中，系统调用作为 C 库的一部分。当我们调用 open 函数的时候流程如图 5.1.2 所示：

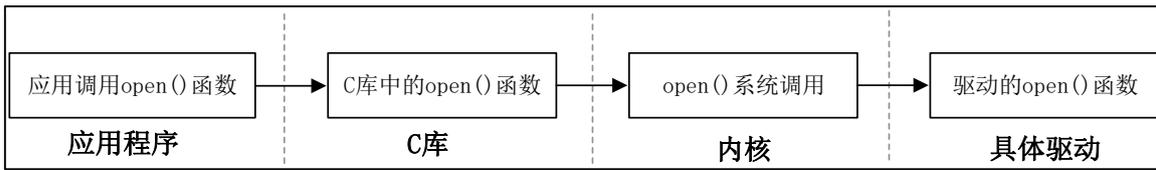


图 5.1.2 open 函数调用流程

其中关于 C 库以及如何通过系统调用“陷入”到内核空间这个我们不用去管，我们重点关注的是应用程序和具体的驱动，应用程序使用到的函数在具体驱动程序中都有与之对应的函数，比如应用程序中调用了 open 这个函数，那么在驱动程序中也得有一个名为 open 的函数。每一个系统调用，在驱动中都有与之对应的一个驱动函数，在 Linux 内核文件 include/linux/fs.h 中有个叫做 file_operations 的结构体，此结构体就是 Linux 内核驱动操作函数集合，内容如下所示：

示例代码 5.1.1 file_operations 结构体

```

1770 struct file_operations {
1771     struct module *owner;
1772     loff_t (*llseek) (struct file *, loff_t, int);
1773     ssize_t (*read) (struct file *, char __user *, size_t,
1774                     loff_t *);
1774     ssize_t (*write) (struct file *, const char __user *, size_t,
1775                      loff_t *);
1775     ssize_t (*read_iter) (struct kiocb *, struct iov_iter *);
1776     ssize_t (*write_iter) (struct kiocb *, struct iov_iter *);
1777     int (*iterate) (struct file *, struct dir_context *);
1778     int (*iterate_shared) (struct file *, struct dir_context *);
1779     __poll_t (*poll) (struct file *, struct poll_table_struct *);
1780     long (*unlocked_ioctl) (struct file *, unsigned int,
1781                             unsigned long);
1781     long (*compat_ioctl) (struct file *, unsigned int,
1782                            unsigned long);
1782     int (*mmap) (struct file *, struct vm_area_struct *);
1783     unsigned long mmap_supported_flags;
1784     int (*open) (struct inode *, struct file *);
1785     int (*flush) (struct file *, fl_owner_t id);
1786     int (*release) (struct inode *, struct file *);
1787     int (*fsync) (struct file *, loff_t, loff_t, int datasync);
1788     int (*fasync) (int, struct file *, int);
1789     int (*lock) (struct file *, int, struct file_lock *);
1790     ssize_t (*sendpage) (struct file *, struct page *, int, size_t,
1791                          loff_t *, int);
1791     unsigned long (*get_unmapped_area) (struct file *, unsigned long,
1792                                         unsigned long, unsigned long);
1792     int (*check_flags) (int);
1793     int (*flock) (struct file *, int, struct file_lock *);
  
```

```

1794     ssize_t (*splice_write)(struct pipe_inode_info *, struct file *,
                             loff_t *, size_t, unsigned int);
1795     ssize_t (*splice_read)(struct file *, loff_t *,
                             struct pipe_inode_info *, size_t, unsigned int);
1796     int (*setlease)(struct file *, long, struct file_lock **,
                     void **);
1797     long (*fallocate)(struct file *file, int mode, loff_t offset,
1798                      loff_t len);
1799     void (*show_fdinfo)(struct seq_file *m, struct file *f);
1800 #ifndef CONFIG_MMU
1801     unsigned (*mmap_capabilities)(struct file *);
1802 #endif
1803     ssize_t (*copy_file_range)(struct file *, loff_t, struct file *,
1804                               loff_t, size_t, unsigned int);
1805     int (*clone_file_range)(struct file *, loff_t, struct file *,
1806                             loff_t, u64);
1807     int (*dedupe_file_range)(struct file *, loff_t, struct file *,
1808                              loff_t,
1809                              u64);
1809     int (*fadvise)(struct file *, loff_t, loff_t, int);
1810 } __randomize_layout;
    
```

简单介绍一下 `file_operation` 结构体中比较重要的、常用的函数:

第 1771 行, `owner` 拥有该结构体的模块的指针, 一般设置为 `THIS_MODULE`。

第 1772 行, `llseek` 函数用于修改文件当前的读写位置。

第 1773 行, `read` 函数用于读取设备文件。

第 1774 行, `write` 函数用于向设备文件写入(发送)数据。

第 1779 行, `poll` 是个轮询函数, 用于查询设备是否可以进行非阻塞的读写。

第 1780 行, `unlocked_ioctl` 函数提供对于设备的控制功能, 与应用程序中的 `ioctl` 函数对应。

第 1781 行, `compat_ioctl` 函数与 `unlocked_ioctl` 函数功能一样, 区别在于在 64 位系统上, 32 位的应用程序调用将会使用此函数。在 32 位的系统上运行 32 位的应用程序调用的是 `unlocked_ioctl`。

第 1782 行, `mmap` 函数用于将设备的内存映射到进程空间中(也就是用户空间), 一般帧缓冲设备会使用此函数, 比如 LCD 驱动的显存, 将帧缓冲(LCD 显存)映射到用户空间中以后应用程序就可以直接操作显存了, 这样就不用用户在用户空间和内核空间之间来回复制。

第 1784 行, `open` 函数用于打开设备文件。

第 1786 行, `release` 函数用于释放(关闭)设备文件, 与应用程序中的 `close` 函数对应。

第 1788 行, `fsync` 函数用于刷新待处理的数据, 用于将缓冲区中的数据刷新到磁盘中。

在字符设备驱动开发中最常用的就是上面这些函数, 关于其他的函数大家可以查阅相关文档。我们在字符设备驱动开发中最主要的工作就是实现上面这些函数, 不一定全部都要实现, 但是像 `open`、`release`、`write`、`read` 等都是需要实现的, 当然了, 具体需要实现哪些函数还是要看具体的驱动要求。

5.2 字符设备驱动开发步骤

上一小节我们简单的介绍了一下字符设备驱动，那么字符设备驱动开发都有哪些步骤呢？我们在学习裸机或者 STM32 的时候关于驱动的开发就是初始化相应的外设寄存器，在 Linux 驱动开发中肯定也是要初始化相应的外设寄存器，这个是毫无疑问的。只是在 Linux 驱动开发中我们需要按照其规定的框架来编写驱动，所以说学 Linux 驱动开发重点是学习其驱动框架。

5.2.1 驱动模块的加载和卸载

Linux 驱动有两种运行方式，第一种就是将驱动编译进 Linux 内核中，这样当 Linux 内核启动的时候就会自动运行驱动程序。第二种就是将驱动编译成模块(Linux 下模块扩展名为.ko)，在 Linux 内核启动以后使用“modprobe”或者“insmod”命令加载驱动模块，本教程我们统一使用“modprobe”命令。在调试驱动的时候一般都选择将其编译为模块，这样我们修改驱动以后只需要编译一下驱动代码即可，不需要编译整个 Linux 代码。而且在调试的时候只需要加载或者卸载驱动模块即可，不需要重启整个系统。总之，将驱动编译为模块最大的好处就是方便开发，当驱动开发完成，确定没有问题以后就可以将驱动编译进 Linux 内核中，当然也可以不编译进 Linux 内核中，具体看自己的需求。

模块有加载和卸载两种操作，我们在编写驱动的时候需要注册这两种操作函数，模块的加载和卸载注册函数如下：

```
module_init(xxx_init);    //注册模块加载函数
module_exit(xxx_exit);    //注册模块卸载函数
```

module_init 函数用来向 Linux 内核注册一个模块加载函数，参数 xxx_init 就是需要注册的具体函数，当使用“modprobe”命令加载驱动的时候，xxx_init 这个函数就会被调用。module_exit 函数用来向 Linux 内核注册一个模块卸载函数，参数 xxx_exit 就是需要注册的具体函数，当使用“rmmod”命令卸载具体驱动的时候 xxx_exit 函数就会被调用。字符设备驱动模块加载和卸载模板如下所示：

示例代码 5.2.1.1 字符设备驱动模块加载和卸载函数模板

```
1  /* 驱动入口函数 */
2  static int __init xxx_init(void)
3  {
4      /* 入口函数具体内容 */
5      return 0;
6  }
7
8  /* 驱动出口函数 */
9  static void __exit xxx_exit(void)
10 {
11     /* 出口函数具体内容 */
12 }
13
14 /* 将上面两个函数指定为驱动的入口和出口函数 */
15 module_init(xxx_init);
16 module_exit(xxx_exit);
```

第 2 行，定义了个名为 xxx_init 的驱动入口函数，并且使用了“__init”来修饰。

第 9 行, 定义了个名为 `xxx_exit` 的驱动出口函数, 并且使用了 “`__exit`” 来修饰。

第 15 行, 调用函数 `module_init` 来声明 `xxx_init` 为驱动入口函数, 当加载驱动的时候 `xxx_init` 函数就会被调用。

第 16 行, 调用函数 `module_exit` 来声明 `xxx_exit` 为驱动出口函数, 当卸载驱动的时候 `xxx_exit` 函数就会被调用。

驱动编译完成以后扩展名为 `.ko`, 前面说了, 有两种命令可以加载驱动模块: `insmod` 和 `modprobe`, `insmod` 是最简单的模块加载命令, 此命令用于加载指定的 `.ko` 模块, 比如加载 `drv.ko` 这个驱动模块, 命令如下:

```
insmod drv.ko
```

`insmod` 命令不能解决模块的依赖关系, 比如 `drv.ko` 依赖 `first.ko` 这个模块, 就必须先使用 `insmod` 命令加载 `first.ko` 这个模块, 然后再加载 `drv.ko` 这个模块。但是 `modprobe` 就不会存在这个问题, `modprobe` 会分析模块的依赖关系, 然后将所有的依赖模块都加载到内核中, 因此 `modprobe` 命令相比 `insmod` 要智能一些。`modprobe` 命令主要智能在提供了模块的依赖性分析、错误检查、错误报告等功能, 推荐使用 `modprobe` 命令来加载驱动。`modprobe` 命令默认会去 `/lib/modules/<kernel-version>` 目录中查找模块, 比如本书使用的 Linux kernel 的版本号为 4.19.232, 因此 `modprobe` 命令默认会到 `/lib/modules/4.19.232` 这个目录中查找相应的驱动模块, 一般自己制作的根文件系统中是不会有这个目录的, 所以需要自己手动创建。

驱动模块的卸载使用命令 “`rmmod`” 即可, 比如要卸载 `drv.ko`, 使用如下命令即可:

```
rmmod drv.ko
```

也可以使用 “`modprobe -r`” 命令卸载驱动, 比如要卸载 `drv.ko`, 命令如下:

```
modprobe -r drv
```

使用 `modprobe` 命令可以卸载掉驱动模块所依赖的其他模块, 前提是这些依赖模块已经没有被其他模块所使用, 否则就不能使用 `modprobe` 来卸载驱动模块。所以对于模块的卸载, 还是推荐使用 `rmmod` 命令。

5.2.2 字符设备注册与注销

对于字符设备驱动而言, 当驱动模块加载成功以后需要注册字符设备, 同样, 卸载驱动模块的时候也需要注销掉字符设备。字符设备的注册和注销函数原型如下所示:

```
static inline int register_chrdev(unsigned int      major,
                                const char        *name,
                                const struct file_operations *fops)

static inline void unregister_chrdev(unsigned int  major,
                                     const char    *name)
```

`register_chrdev` 函数用于注册字符设备, 此函数一共有三个参数, 这三个参数的含义如下:
major: 主设备号, Linux 下每个设备都有一个设备号, 设备号分为主设备号和次设备号两部分, 关于设备号后面会详细讲解。

name: 设备名字, 指向一串字符串。

fops: 结构体 `file_operations` 类型指针, 指向设备的操作函数集合变量。

`unregister_chrdev` 函数用于注销字符设备, 此函数有两个参数, 这两个参数含义如下:

major: 要注销的设备对应的主设备号。

name: 要注销的设备对应的设备名。

一般字符设备的注册在驱动模块的入口函数 `xxx_init` 中进行，字符设备的注销在驱动模块的出口函数 `xxx_exit` 中进行。在示例代码 5.2.1.1 中字符设备的注册和注销，内容如下所示：

示例代码 5.2.2.1 加入字符设备注册和注销

```

1  static struct file_operations test_fops;
2
3  /* 驱动入口函数 */
4  static int __init xxx_init(void)
5  {
6      /* 入口函数具体内容 */
7      int retvalue = 0;
8
9      /* 注册字符设备驱动 */
10     retvalue = register_chrdev(200, "chrtest", &test_fops);
11     if(retvalue < 0){
12         /* 字符设备注册失败,自行处理 */
13     }
14     return 0;
15 }
16
17 /* 驱动出口函数 */
18 static void __exit xxx_exit(void)
19 {
20     /* 注销字符设备驱动 */
21     unregister_chrdev(200, "chrtest");
22 }
23
24 /* 将上面两个函数指定为驱动的入口和出口函数 */
25 module_init(xxx_init);
26 module_exit(xxx_exit);
    
```

第 1 行，定义了一个 `file_operations` 结构体变量 `test_fops`，`test_fops` 就是设备的操作函数集合，只是此时我们还没有初始化 `test_fops` 中的 `open`、`release` 等这些成员变量，所以这个操作函数集合还是空的。

第 10 行，调用函数 `register_chrdev` 注册字符设备，主设备号为 200，设备名字为“chrtest”，设备操作函数集合就是第 1 行定义的 `test_fops`。要注意的一点就是，选择没有被使用的主设备号，输入命令“`cat /proc/devices`”可以查看当前已经被使用掉的设备号，如图 5.2.2.1 所示(限于篇幅原因，只展示一部分)：

```

root@ATK-DLRK356X:/# cat /proc/devices
Character devices:
 1 mem
 4 /dev/vc/0
 4 tty
 4 ttyS
 5 /dev/tty
 5 /dev/console
 5 /dev/ptmx
 7 vcs
10 misc
13 input
29 fb
81 video4linux
89 i2c
90 mtd
108 ppp
    
```

图 5.2.2.1 查看当前设备

在图 5.2.2.1 中可以列出当前系统中所有的字符设备和块设备，其中第 1 列就是设备对应的主设备号。200 这个主设备号在我的开发板中并没有被使用，所以我这里就用了 200 这个主设备号。

第 21 行，调用函数 `unregister_chrdev` 注销主设备号为 200 的这个设备。

5.2.3 实现设备的具体操作函数

`file_operations` 结构体就是设备的具体操作函数，在示例代码 5.2.2.1 中我们定义了 `file_operations` 结构体类型的变量 `test_fops`，但是还没对其进行初始化，也就是初始化其中的 `open`、`release`、`read` 和 `write` 等具体的设备操作函数。本节我们就完成变量 `test_fops` 的初始化，设置好针对 `chrtest` 设备的操作函数。在初始化 `test_fops` 之前我们要分析一下需求，也就是要对 `chrtest` 这个设备进行哪些操作，只有确定了需求以后才知道我们应该实现哪些操作函数。假设对 `chrtest` 这个设备有如下两个要求：

1、能够对 `chrtest` 进行打开和关闭操作

设备打开和关闭是最基本的要求，几乎所有的设备都得提供打开和关闭的功能。因此我们需要实现 `file_operations` 中的 `open` 和 `release` 这两个函数。

2、对 `chrtest` 进行读写操作

假设 `chrtest` 这个设备控制着一段缓冲区(内存)，应用程序需要通过 `read` 和 `write` 这两个函数对 `chrtest` 的缓冲区进行读写操作。所以需要实现 `file_operations` 中的 `read` 和 `write` 这两个函数。

需求很清晰了，修改示例代码 5.2.2.1，在其中加入 `test_fops` 这个结构体变量的初始化操作，完成以后的内容如下所示：

示例代码 5.2.3.1 加入设备操作函数

```

1  /* 打开设备 */
2  static int chrtest_open(struct inode *inode, struct file *filp)
3  {
4      /* 用户实现具体功能 */
5      return 0;
6  }
7
8  /* 从设备读取 */
    
```

```

9  static ssize_t chrtest_read(struct file *filp, char __user *buf,
                               size_t cnt, loff_t *offt)
10 {
11     /* 用户实现具体功能 */
12     return 0;
13 }
14
15 /* 向设备写数据 */
16 static ssize_t chrtest_write(struct file *filp,
                               const char __user *buf,
                               size_t cnt, loff_t *offt)
17 {
18     /* 用户实现具体功能 */
19     return 0;
20 }
21
22 /* 关闭/释放设备 */
23 static int chrtest_release(struct inode *inode, struct file *filp)
24 {
25     /* 用户实现具体功能 */
26     return 0;
27 }
28
29 static struct file_operations test_fops = {
30     .owner = THIS_MODULE,
31     .open = chrtest_open,
32     .read = chrtest_read,
33     .write = chrtest_write,
34     .release = chrtest_release,
35 };
36
37 /* 驱动入口函数 */
38 static int __init xxx_init(void)
39 {
40     /* 入口函数具体内容 */
41     int retvalue = 0;
42
43     /* 注册字符设备驱动 */
44     retvalue = register_chrdev(200, "chrtest", &test_fops);
45     if(retvalue < 0){
46         /* 字符设备注册失败,自行处理 */
47     }
48     return 0;

```

```

49 }
50
51 /* 驱动出口函数 */
52 static void __exit xxx_exit(void)
53 {
54     /* 注销字符设备驱动 */
55     unregister_chrdev(200, "chrtest");
56 }
57
58 /* 将上面两个函数指定为驱动的入口和出口函数 */
59 module_init(xxx_init);
60 module_exit(xxx_exit);
    
```

在示例代码 5.2.3.1 中我们一开始编写了四个函数: chrtest_open、chrtest_read、chrtest_write 和 chrtest_release。这四个函数就是 chrtest 设备的 open、read、write 和 release 操作函数。第 29 行~35 行初始化 test_fops 的 open、read、write 和 release 这四个成员变量。

5.2.4 添加 LICENSE 和作者信息

最后我们需要在驱动中加入 LICENSE 信息和作者信息, 其中 LICENSE 是必须添加的, 否则的话编译的时候会报错, 作者信息可以添加也可以不添加。LICENSE 和作者信息的添加使用如下两个函数:

```

MODULE_LICENSE()    //添加模块 LICENSE 信息
MODULE_AUTHOR()    //添加模块作者信息
    
```

最后给示例代码 5.2.3.1 加入 LICENSE 和作者信息, 完成以后的内容如下:

示例代码 5.2.4.1 字符设备驱动最终的模板

```

1 /* 打开设备 */
2 static int chrtest_open(struct inode *inode, struct file *filp)
3 {
4     /* 用户实现具体功能 */
5     return 0;
6 }
.....
57
58 /* 将上面两个函数指定为驱动的入口和出口函数 */
59 module_init(xxx_init);
60 module_exit(xxx_exit);
61
62 MODULE_LICENSE("GPL");
63 MODULE_AUTHOR("alientek");
    
```

第 62 行, LICENSE 采用 GPL 协议。

第 63 行, 添加作者名字。

至此, 字符设备驱动开发的完整步骤就讲解完了, 而且也编写好了一个完整的字符设备驱动模板, 以后字符设备驱动开发都可以在此模板上进行。

5.3 Linux 设备号

5.3.1 设备号的组成

为了方便管理, Linux 中每个设备都有一个设备号, 设备号由主设备号和次设备号两部分组成, 主设备号表示某一个具体的驱动, 次设备号表示使用这个驱动的各个设备。Linux 提供了一个名为 `dev_t` 的数据类型表示设备号, `dev_t` 定义在文件 `include/linux/types.h` 里面, 定义如下:

示例代码 5.3.1.1 设备号 `dev_t`

```
13 typedef u32 __kernel_dev_t;
.....
16 typedef __kernel_dev_t dev_t;
```

可以看出 `dev_t` 是 `u32` 类型的, 也就是 `unsigned int`, 所以 `dev_t` 其实就是 `unsigned int` 类型, 是一个 32 位的数据类型。这 32 位的数据构成了主设备号和次设备号两部分, 其中高 12 位为主设备号, 低 20 位为次设备号。因此 Linux 系统中主设备号范围为 0~4095, 所以大家在选择主设备号的时候一定不要超过这个范围。在文件 `include/linux/kdev_t.h` 中提供了几个关于设备号的操作函数(本质是宏), 如下所示:

示例代码 5.3.1.2 设备号操作函数

```
7 #define MINORBITS 20
8 #define MINORMASK ((1U << MINORBITS) - 1)
9
10 #define MAJOR(dev) ((unsigned int) ((dev) >> MINORBITS))
11 #define MINOR(dev) ((unsigned int) ((dev) & MINORMASK))
12 #define MKDEV(ma,mi) (((ma) << MINORBITS) | (mi))
```

第 7 行, 宏 `MINORBITS` 表示次设备号位数, 一共是 20 位。

第 8 行, 宏 `MINORMASK` 表示次设备号掩码。

第 10 行, 宏 `MAJOR` 用于从 `dev_t` 中获取主设备号, 将 `dev_t` 右移 20 位即可。

第 11 行, 宏 `MINOR` 用于从 `dev_t` 中获取次设备号, 取 `dev_t` 的低 20 位的值即可。

第 12 行, 宏 `MKDEV` 用于将给定的主设备号和次设备号的值组合成 `dev_t` 类型的设备号。

5.3.2 设备号的分配

1、静态分配设备号

本小节讲的设备号分配主要是主设备号的分配。前面讲解字符设备驱动的时候说过了, 注册字符设备的时候需要给设备指定一个设备号, 这个设备号可以是驱动开发者静态指定的一个设备号, 比如 200 这个主设备号。有一些常用的设备号已经被 Linux 内核开发者给分配掉了, 具体分配的内容可以查看文档 `Documentation/devices.txt`。并不是说内核开发者已经分配掉的主设备号我们就不能用了, 具体能不能用还得看我们的硬件平台运行过程中有没有使用这个主设备号, 使用 “`cat /proc/devices`” 命令即可查看当前系统中所有已经使用了的设备号。

2、动态分配设备号

静态分配设备号需要我们检查当前系统中所有被使用了的设备号, 然后挑选一个没有使用的。而且静态分配设备号很容易带来冲突问题, Linux 社区推荐使用动态分配设备号, 在注册字符设备之前先申请一个设备号, 系统会自动给你一个没有被使用的设备号, 这样就避免了冲突。卸载驱动的时候释放掉这个设备号即可, 设备号的申请函数如下:

```
int alloc_chrdev_region(dev_t *dev, unsigned baseminor, unsigned count, const char *name)
```

函数 alloc_chrdev_region 用于申请设备号，此函数有 4 个参数：

dev: 保存申请到的设备号。

baseminor: 次设备号起始地址，alloc_chrdev_region 可以申请一段连续的多个设备号，这些设备号的主设备号一样，但是次设备号不同，次设备号以 baseminor 为起始地址地址开始递增。一般 baseminor 为 0，也就是说次设备号从 0 开始。

count: 要申请的设备号数量。

name: 设备名字。

注销字符设备之后要释放掉设备号，设备号释放函数如下：

```
void unregister_chrdev_region(dev_t from, unsigned count)
```

此函数有两个参数：

from: 要释放的设备号。

count: 表示从 from 开始，要释放的设备号数量。

5.4 chrdevbase 字符设备驱动开发实验

字符设备驱动开发的基本步骤我们已经了解了，本节我们就以 chrdevbase 这个虚拟设备为例，完整的编写一个字符设备驱动模块。chrdevbase 不是实际存在的一个设备，是笔者为了方便讲解字符设备的开发而引入的一个虚拟设备。chrdevbase 设备有两个缓冲区，一个读缓冲区，一个写缓冲区，这两个缓冲区的大小都为 100 字节。在应用程序中可以向 chrdevbase 设备的写缓冲区中写入数据，从读缓冲区中读取数据。chrdevbase 这个虚拟设备的功能很简单，但是它包含了字符设备的最基本功能。

5.4.1 实验程序编写

本实验对应的例程路径为：[开发板光盘](#) → [Linux 驱动例程源码](#) → [01_chrdevbase](#)。

应用程序调用 open 函数打开 chrdevbase 这个设备，打开以后可以使用 write 函数向 chrdevbase 的写缓冲区 writebuf 中写入数据(不超过 100 个字节)，也可以使用 read 函数读取读缓冲区 readbuf 中的数据操作，操作完成以后应用程序使用 close 函数关闭 chrdevbase 设备。

1、创建 VSCode 工程

在 Ubuntu 中创建一个目录用来存放 Linux 驱动程序，比如我创建了一个名为 Linux_Drivers 的目录来存放所有的 Linux 驱动。在 Linux_Drivers 目录下新建一个名为 01_chrdevbase 的子目录来存放本实验所有文件，如图 5.4.1.1 所示：

```
alientek@ubuntu:~/Linux_Drivers$ ls
01_chrdevbase
alientek@ubuntu:~/Linux_Drivers$
```

图 5.4.1.1 Linux 实验程序目录

在 01_chrdevbase 目录中新建 VSCode 工程或者将工作区另存到 01_chrdevbase 目录下，并且新建 chrdevbase.c 文件，完成以后 1_chrdevbase 目录中的文件如图 5.4.1.2 所示：

```
alientek@ubuntu:~/Linux_Drivers/01_chrdevbase$ ls -a
.  ..  01_chrdevbase.code-workspace  chrdevbase.c
alientek@ubuntu:~/Linux_Drivers/01_chrdevbase$
```

图 5.4.1.2 01_chrdevbase 目录文件

2、添加头文件路径

因为是编写 Linux 驱动,因此会用到 Linux 源码中的函数。我们需要在 VSCode 中添加 Linux 源码中的头文件路径。打开 VSCode,按下“Ctrl+Shift+P”打开 VSCode 的控制台,然后输入“C/C++: Edit configurations(JSON)”,打开 C/C++编辑配置文件,如图 5.4.1.3 所示:



图 5.4.1.3 C/C++编辑配置文件

打开以后会自动在.vscode 目录下生成一个名为 c_cpp_properties.json 的文件,此文件默认内容如下所示:

示例代码 5.4.1.1 c_cpp_properties.json 文件原内容

```

1 {
2   "configurations": [
3     {
4       "name": "Linux",
5       "includePath": [
6         "${workspaceFolder}/**",
7       ],
8       "defines": [],
9       "compilerPath": "/usr/bin/clang",
10      "cStandard": "c11",
11      "cppStandard": "c++17",
12      "intelliSenseMode": "clang-x64"
13    }
14  ],
15  "version": 4
16 }

```

第 5 行的 includePath 表示头文件路径,需要将 Linux 源码里面的头文件路径添加进来,添加头文件路径以后的 c_cpp_properties.json 的文件内容如下所示:(注意 generated 文件夹必须先编译内核成功才会生成,并且需要确认自己的 SDK 路径)

示例代码 5.4.1.2 添加头文件路径后的 c_cpp_properties.json

```

1 {
2   "configurations": [
3     {
4       "name": "Linux",
5       "includePath": [
6         "${workspaceFolder}/**",
7         "/home/alientek/rk3568_linux_sdk/kernel/
8         arch/arm64/include",
9         "/home/alientek/rk3568_linux_sdk/kernel/include",
10        "/home/alientek/rk3568_linux_sdk/kernel/
11        arch/arm64/include/generated "
12      ],
13      "defines": [],

```

```

12     "compilerPath": "/usr/bin/gcc",
13     "cStandard": "gnu11",
14     "cppStandard": "gnu++14",
15     "intelliSenseMode": "gcc-x64"
16     }
17 ],
18     "version": 4
19 }
    
```

第7~9行就是添加好的Linux头文件路径。分别是开发板所使用的Linux源码下的include、arch/arm64/include和arch/arm64/include/generated这三个目录的路径，注意，这里使用了绝对路径。

3、编写实验程序

工程建立好以后就可以开始编写驱动程序了，新建chrdevbase.c，然后在里面输入如下内容：

示例代码 5.4.1.3 chrdevbase.c 文件

```

1  #include <linux/types.h>
2  #include <linux/kernel.h>
3  #include <linux/delay.h>
4  #include <linux/ide.h>
5  #include <linux/init.h>
6  #include <linux/module.h>
7  /*****
8  Copyright © ALIENTEK Co., Ltd. 1998-2029. All rights reserved.
9  文件名   : chrdevbase.c
10  作者    : 正点原子
11  版本    : V1.0
12  描述    : chrdevbase 驱动文件。
13  其他    : 无
14  论坛    : www.openedv.com
15  日志    : 初版 V1.0 2022/12/01 正点原子团队创建
16  *****/
17
18  #define CHRDEVBASE_MAJOR    200                /* 主设备号    */
19  #define CHRDEVBASE_NAME    "chrdevbase"        /* 设备名      */
20
21  static char readbuf[100];                      /* 读缓冲区    */
22  static char writebuf[100];                    /* 写缓冲区    */
23  static char kerneldata[] = {"kernel data!"};
24
25  /*
26  * @description   : 打开设备
27  * @param - inode : 传递给驱动的 inode
28  * @param - filp  : 设备文件, file 结构体有个叫做 private_data 的成员变量
29  *                一般在 open 的时候将 private_data 指向设备结构体。
    
```

```

30 * @return      : 0 成功;其他 失败
31 */
32 static int chrdevbase_open(struct inode *inode, struct file *filp)
33 {
34     //printk("chrdevbase open!\r\n");
35     return 0;
36 }
37
38 /*
39 * @description  : 从设备读取数据
40 * @param - filp : 要打开的设备文件(文件描述符)
41 * @param - buf  : 返回给用户空间的数据缓冲区
42 * @param - cnt  : 要读取的数据长度
43 * @param - offt : 相对于文件首地址的偏移
44 * @return      : 读取的字节数, 如果为负值, 表示读取失败
45 */
46 static ssize_t chrdevbase_read(struct file *filp, char __user *buf,
                                size_t cnt, loff_t *offt)
47 {
48     int retvalue = 0;
49
50     /* 向用户空间发送数据 */
51     memcpy(readbuf, kerneldata, sizeof(kerneldata));
52     retvalue = copy_to_user(buf, readbuf, cnt);
53     if(retvalue == 0){
54         printk("kernel senddata ok!\r\n");
55     }else{
56         printk("kernel senddata failed!\r\n");
57     }
58
59     //printk("chrdevbase read!\r\n");
60     return 0;
61 }
62
63 /*
64 * @description  : 向设备写数据
65 * @param - filp : 设备文件, 表示打开的文件描述符
66 * @param - buf  : 要写给设备写入的数据
67 * @param - cnt  : 要写入的数据长度
68 * @param - offt : 相对于文件首地址的偏移
69 * @return      : 写入的字节数, 如果为负值, 表示写入失败
70 */
71 static ssize_t chrdevbase_write(struct file *filp,

```

```

const char __user *buf,
size_t cnt, loff_t *offt)
72 {
73     int retvalue = 0;
74     /* 接收用户空间传递给内核的数据并且打印出来 */
75     retvalue = copy_from_user(writebuf, buf, cnt);
76     if(retvalue == 0){
77         printk("kernel recevdata:%s\r\n", writebuf);
78     }else{
79         printk("kernel recevdata failed!\r\n");
80     }
81
82     //printk("chrdevbase write!\r\n");
83     return 0;
84 }
85
86 /*
87  * @description   : 关闭/释放设备
88  * @param - filp  : 要关闭的设备文件(文件描述符)
89  * @return        : 0 成功;其他 失败
90  */
91 static int chrdevbase_release(struct inode *inode,
                               struct file *filp)
92 {
93     //printk("chrdevbase release! \r\n");
94     return 0;
95 }
96
97 /*
98  * 设备操作函数结构体
99  */
100 static struct file_operations chrdevbase_fops = {
101     .owner = THIS_MODULE,
102     .open = chrdevbase_open,
103     .read = chrdevbase_read,
104     .write = chrdevbase_write,
105     .release = chrdevbase_release,
106 };
107
108 /*
109  * @description   : 驱动入口函数
110  * @param        : 无
111  * @return        : 0 成功;其他 失败

```

```

112 */
113 static int __init chrdevbase_init(void)
114 {
115     int retvalue = 0;
116
117     /* 注册字符设备驱动 */
118     retvalue = register_chrdev(CHRDEVBASE_MAJOR, CHRDEVBASE_NAME,
119                               &chrdevbase_fops);
119     if(retvalue < 0){
120         printk("chrdevbase driver register failed\r\n");
121     }
122     printk("chrdevbase_init()\r\n");
123     return 0;
124 }
125
126 /*
127 * @description   : 驱动出口函数
128 * @param         : 无
129 * @return        : 无
130 */
131 static void __exit chrdevbase_exit(void)
132 {
133     /* 注销字符设备驱动 */
134     unregister_chrdev(CHRDEVBASE_MAJOR, CHRDEVBASE_NAME);
135     printk("chrdevbase_exit()\r\n");
136 }
137
138 /*
139 * 将上面两个函数指定为驱动的入口和出口函数
140 */
141 module_init(chrdevbase_init);
142 module_exit(chrdevbase_exit);
143
144 /*
145 * LICENSE 和作者信息
146 */
147 MODULE_LICENSE("GPL");
148 MODULE_AUTHOR("ALIENTEK");
149 MODULE_INFO(intree, "Y");
    
```

第 32~36 行, `chrdevbase_open` 函数, 当应用程序调用 `open` 函数的时候此函数就会调用。本例程中我们没有做任何工作, 只是输出一串字符, 用于调试。这里使用了 `printk` 来输出信息, 而不是 `printf`! 因为在 Linux 内核中没有 `printf` 这个函数。`printk` 相当于 `printf` 的孪生兄妹, `printf` 运行在用户态, `printk` 运行在内核态。在内核中想要向控制台输出或显示一些内容, 必须使用

printk 这个函数。不同之处在于，printk 可以根据日志级别对消息进行分类，一共有 8 个消息级别，这 8 个消息级别定义在文件 include/linux/kern_levels.h 里面，定义如下：

```
#define KERN_SOH          "\001"
#define KERN_EMERG       KERN_SOH "0" /* 紧急事件，一般是内核崩溃 */
#define KERN_ALERT       KERN_SOH "1" /* 必须立即采取行动 */
#define KERN_CRIT        KERN_SOH "2" /* 临界条件，比如严重的软件或硬件错误*/
#define KERN_ERR         KERN_SOH "3" /* 错误状态，一般设备驱动程序中使用
                                   KERN_ERR 报告硬件错误 */
#define KERN_WARNING     KERN_SOH "4" /* 警告信息，不会对系统造成严重影响 */
#define KERN_NOTICE      KERN_SOH "5" /* 有必要进行提示的一些信息 */
#define KERN_INFO        KERN_SOH "6" /* 提示性的信息 */
#define KERN_DEBUG       KERN_SOH "7" /* 调试信息 */
```

一共定义了 8 个级别，其中 0 的优先级最高，7 的优先级最低。如果要设置消息级别，参考如下示例：

```
printk(KERN_EMERG "gsmi: Log Shutdown Reason\n");
```

上述代码就是设置“gsmi: Log Shutdown Reason\n”这行消息的级别为 KERN_EMERG。在具体的消息前面加上 KERN_EMERG 就可以将这条消息的级别设置为 KERN_EMERG。如果使用 printk 的时候不显式的设置消息级别，那么 printk 将会采用默认级别 MESSAGE_LOGLEVEL_DEFAULT。

在 include/linux/printk.h 中有个宏 CONSOLE_LOGLEVEL_DEFAULT，定义如下：

```
#define CONSOLE_LOGLEVEL_DEFAULT CONFIG_CONSOLE_LOGLEVEL_DEFAULT
MESSAGE_LOGLEVEL_DEFAULT 和 CONFIG_CONSOLE_LOGLEVEL_DEFAULT 是通
```

过内核图形化界面配置的，配置路径如下：

```
-> Kernel hacking
    -> printk and dmesg options
        -> (7) Default console loglevel (1-15) //设置默认终端消息级别
        -> (4) Default message log level (1-7) //设置默认消息级别
```

“Default console loglevel”就是用来设置 CONSOLE_LOGLEVEL_DEFAULT 的值，“Default message log level”设置 CONFIG_MESSAGE_LOGLEVEL_DEFAULT 的值。默认如图 5.4.1.1 所示：

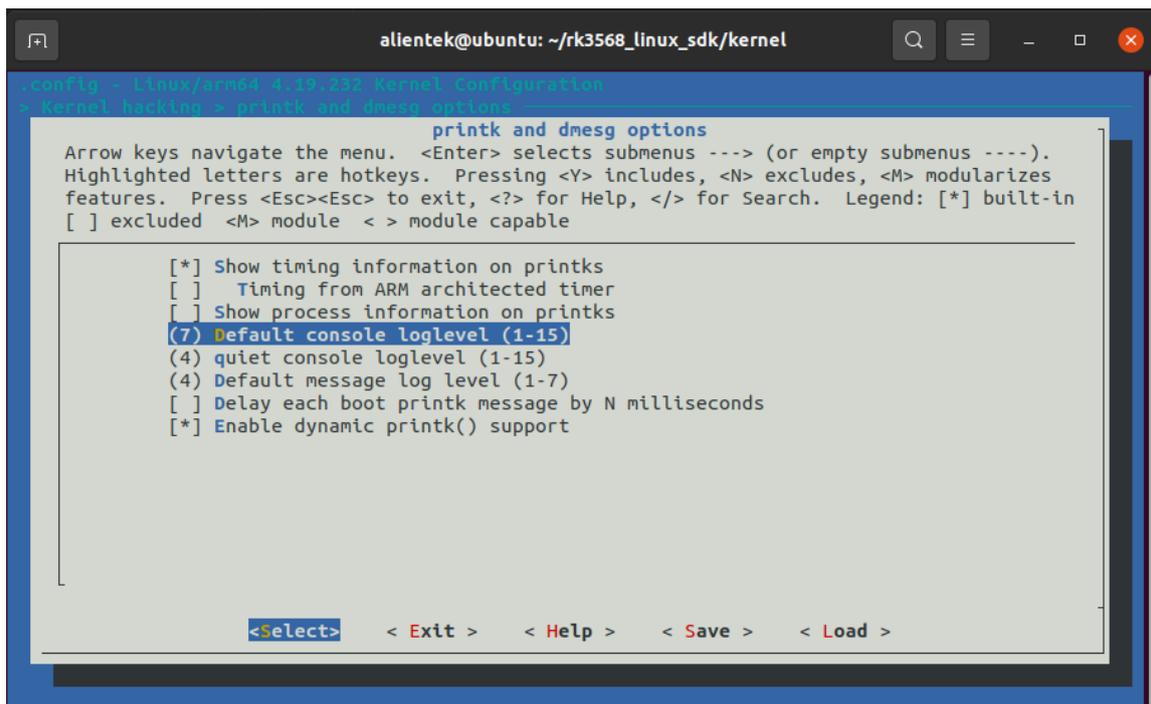


图 5.4.1.1 内核消息配置

图 5.4.1.1 可以看出，默认为 `CONSOLE_LOGLEVEL_DEFAULT` 默认为 7。所以宏 `CONSOLE_LOGLEVEL_DEFAULT` 的值默认也为 7，`MESSAGE_LOGLEVEL_DEFAULT` 默认值为 4。`CONSOLE_LOGLEVEL_DEFAULT` 控制着哪些级别的消息可以显示在控制台上，此宏默认为 7，意味着只有优先级高于 7 的消息才能显示在控制台上。

这个就是 `printk` 和 `printf` 的最大区别，可以通过消息级别来决定哪些消息可以显示在控制台上。默认消息级别为 4，4 的级别比 7 高，所示直接使用 `printk` 输出的信息是可以显示在控制台上的。

参数 `filp` 有个叫做 `private_data` 的成员变量，`private_data` 是个 `void` 指针，一般在驱动中将 `private_data` 指向设备结构体，设备结构体会存放设备的一些属性。

第 46~61 行，`chrdevbase_read` 函数，应用程序调用 `read` 函数从设备中读取数据的时候此函数会执行。参数 `buf` 是用户空间的内存，读取到的数据存储到 `buf` 中，参数 `cnt` 是要读取的字节数，参数 `offt` 是相对于文件首地址的偏移。`kerneldata` 里面保存着用户空间要读取的数据，第 51 行先将 `kerneldata` 数组中的数据拷贝到读缓冲区 `readbuf` 中，第 52 行通过函数 `copy_to_user` 将 `readbuf` 中的数据复制到参数 `buf` 中。因为内核空间不能直接操作用户空间的内存，因此需要借助 `copy_to_user` 函数来完成内核空间的数据到用户空间的复制。`copy_to_user` 函数原型如下：

```
static inline long copy_to_user(void __user *to, const void *from, unsigned long n)
```

参数 `to` 表示目的，参数 `from` 表示源，参数 `n` 表示要复制的数据长度。如果复制成功，返回值为 0，如果复制失败则返回负数。

第 71~84 行，`chrdevbase_write` 函数，应用程序调用 `write` 函数向设备写数据的时候此函数就会执行。参数 `buf` 就是应用程序要写入设备的数据，也是用户空间的内存，参数 `cnt` 是要写入的数据长度，参数 `offt` 是相对文件首地址的偏移。第 75 行通过函数 `copy_from_user` 将 `buf` 中的数据复制到写缓冲区 `writebuf` 中，因为用户空间内存不能直接访问内核空间的内存，所以需要借助函数 `copy_from_user` 将用户空间的数据复制到 `writebuf` 这个内核空间中。

第 91~95 行，`chrdevbase_release` 函数，应用程序调用 `close` 关闭设备文件的时候此函数会执行，一般会在此函数里面执行一些释放操作。如果在 `open` 函数中设置了 `filp` 的 `private_data`

成员变量指向设备结构体, 那么在 `release` 函数最终就要释放掉。

第 100~106 行, 新建 `chrdevbase` 的设备文件操作结构体 `chrdevbase_fops`, 初始化 `chrdevbase_fops`。

第 113~124 行, 驱动入口函数 `chrdevbase_init`, 第 118 行调用函数 `register_chrdev` 来注册字符设备。

第 131~136 行, 驱动出口函数 `chrdevbase_exit`, 第 134 行调用函数 `unregister_chrdev` 来注销字符设备。

第 141~142 行, 通过 `module_init` 和 `module_exit` 这两个函数来指定驱动的入口和出口函数。

第 147~148 行, 添加 `LICENSE` 和作者信息。

第 149 行是为了欺骗内核, 给本驱动添加 `intree` 标记, 如果不加就会有“loading out-of-tree module taints kernel.”这个警告

5.4.2 编写测试 APP

1、C 库文件操作基本函数

编写测试 APP 就是编写 Linux 应用, 需要用到 C 库里面和文件操作有关的一些函数, 比如 `open`、`read`、`write` 和 `close` 这四个函数。

①、open 函数

`open` 函数原型如下:

```
int open(const char *pathname, int flags)
```

`open` 函数参数含义如下:

pathname: 要打开的设备或者文件名。

flags: 文件打开模式, 以下三种模式必选其一:

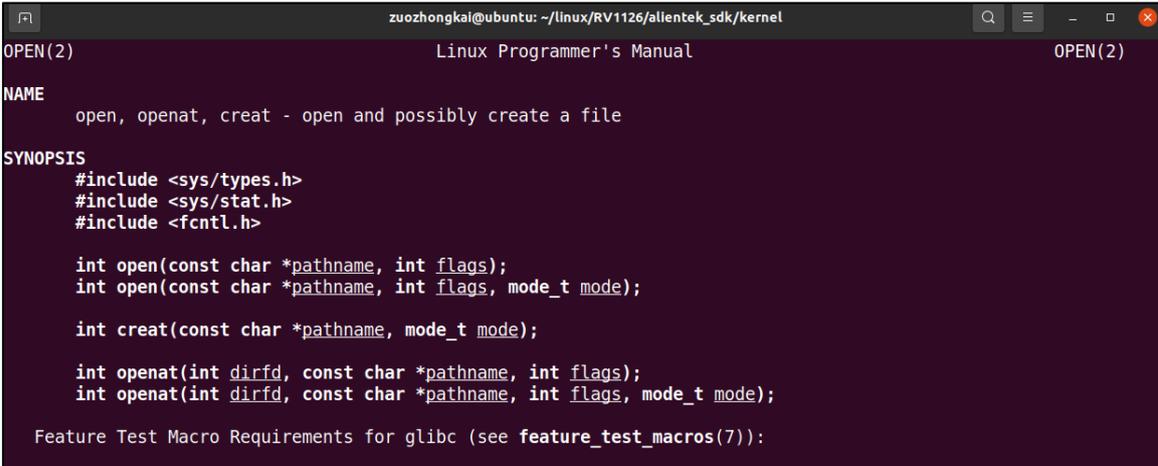
- `O_RDONLY` 只读模式
- `O_WRONLY` 只写模式
- `O_RDWR` 读写模式

因为我们要对 `chrdevbase` 这个设备进行读写操作, 所以选择 `O_RDWR`。除了上述三种模式以外还有其他的可选模式, 通过逻辑或来选择多种模式:

- `O_APPEND` 每次写操作都写入文件的末尾
- `O_CREAT` 如果指定文件不存在, 则创建这个文件
- `O_EXCL` 如果要创建的文件已存在, 则返回 -1, 并且修改 `errno` 的值
- `O_TRUNC` 如果文件存在, 并且以只写/读写方式打开, 则清空文件全部内容
- `O_NOCTTY` 如果路径名指向终端设备, 不要把这个设备用作控制终端。
- `O_NONBLOCK` 如果路径名指向 FIFO/块文件/字符文件, 则把文件的打开和后继 I/O 设置为非阻塞
- `O_DSYNC` 等待物理 I/O 结束后再 `write`。在不影响读取新写入的数据的前提下, 不等待文件属性更新。
- `O_RSYNC` `read` 等待所有写入同一区域的写操作完成后再进行。
- `O_SYNC` 等待物理 I/O 结束后再 `write`, 包括更新文件属性的 I/O。

返回值: 如果文件打开成功的话返回文件的文件描述符。

在 Ubuntu 中输入 “`man 2 open`” 即可查看 `open` 函数的详细内容, 如图 5.4.2.1 所示:



```

zuozhongkai@ubuntu: ~/linux/RV1126/allentek_sdk/kernel
OPEN(2) Linux Programmer's Manual OPEN(2)
NAME
  open, openat, creat - open and possibly create a file
SYNOPSIS
  #include <sys/types.h>
  #include <sys/stat.h>
  #include <fcntl.h>

  int open(const char *pathname, int flags);
  int open(const char *pathname, int flags, mode_t mode);

  int creat(const char *pathname, mode_t mode);

  int openat(int dirfd, const char *pathname, int flags);
  int openat(int dirfd, const char *pathname, int flags, mode_t mode);

  Feature Test Macro Requirements for glibc (see feature_test_macros(7)):
    
```

图 5.4.2.1 open 函数帮助信息

②、read 函数

read 函数原型如下:

```
ssize_t read(int fd, void *buf, size_t count)
```

read 函数参数含义如下:

fd: 要读取的文件描述符, 读取文件之前要先用 open 函数打开文件, open 函数打开文件成功以后会得到文件描述符。

buf: 数据读取到此 buf 中。

count: 要读取的数据长度, 也就是字节数。

返回值: 读取成功的话返回读取到的字节数; 如果返回 0 表示读取到了文件末尾; 如果返回负值, 表示读取失败。在 Ubuntu 中输入 “man 2 read” 命令即可查看 read 函数的详细内容。

③、write 函数

write 函数原型如下:

```
ssize_t write(int fd, const void *buf, size_t count);
```

write 函数参数含义如下:

fd: 要进行写操作的文件描述符, 写文件之前要先用 open 函数打开文件, open 函数打开文件成功以后会得到文件描述符。

buf: 要写入的数据。

count: 要写入的数据长度, 也就是字节数。

返回值: 写入成功的话返回写入的字节数; 如果返回 0 表示没有写入任何数据; 如果返回负值, 表示写入失败。在 Ubuntu 中输入 “man 2 write” 命令即可查看 write 函数的详细内容。

④、close 函数

close 函数原型如下:

```
int close(int fd);
```

close 函数参数含义如下:

fd: 要关闭的文件描述符。

返回值: 0 表示关闭成功, 负值表示关闭失败。在 Ubuntu 中输入 “man 2 close” 命令即可查看 close 函数的详细内容。

2、编写测试 APP 程序

驱动编写好以后是需要测试的, 一般编写一个简单的测试 APP, 测试 APP 运行在用户空间。测试 APP 很简单通过输入相应的指令来对 chrdevbase 设备执行读或者写操作。在

01_chrdevbase 目录中新建 chrdevbaseApp.c 文件, 在此文件中输入如下内容:

示例代码 5.4.2.1 chrdevbaseApp.c 文件

```

1  #include "stdio.h"
2  #include "unistd.h"
3  #include "sys/types.h"
4  #include "sys/stat.h"
5  #include "fcntl.h"
6  #include "stdlib.h"
7  #include "string.h"
8  /*****
9  Copyright © ALIENTEK Co., Ltd. 1998-2029. All rights reserved.
10  文件名      : chrdevbaseApp.c
11  作者       : 正点原子
12  版本       : V1.0
13  描述       : chrdevbase 驱测试 APP。
14  其他       : 使用方法: ./chrdevbaseApp /dev/chrdevbase <1>|<2>
15             argv[2] 1:读文件
16             argv[2] 2:写文件
17 论坛       : www.openedv.com
18 日志       : 初版 V1.0 2022/12/01 正点原子团队创建
19  *****/
20
21 static char usrdata[] = {"usr data!"};
22
23 /*
24  * @description   : main 主程序
25  * @param - argc  : argv 数组元素个数
26  * @param - argv  : 具体参数
27  * @return        : 0 成功;其他 失败
28  */
29 int main(int argc, char *argv[])
30 {
31     int fd, retvalue;
32     char *filename;
33     char readbuf[100], writebuf[100];
34
35     if(argc != 3){
36         printf("Error Usage!\r\n");
37         return -1;
38     }
39
40     filename = argv[1];
41

```

```

42     /* 打开驱动文件 */
43     fd = open(filename, O_RDWR);
44     if(fd < 0){
45         printf("Can't open file %s\r\n", filename);
46         return -1;
47     }
48
49     if(atoi(argv[2]) == 1){ /* 从驱动文件读取数据 */
50         retvalue = read(fd, readbuf, 50);
51         if(retvalue < 0){
52             printf("read file %s failed!\r\n", filename);
53         }else{
54             /* 读取成功, 打印出读取成功的数据 */
55             printf("read data:%s\r\n", readbuf);
56         }
57     }
58
59     if(atoi(argv[2]) == 2){
60         /* 向设备驱动写数据 */
61         memcpy(writebuf, usrdata, sizeof(usrdata));
62         retvalue = write(fd, writebuf, 50);
63         if(retvalue < 0){
64             printf("write file %s failed!\r\n", filename);
65         }
66     }
67
68     /* 关闭设备 */
69     retvalue = close(fd);
70     if(retvalue < 0){
71         printf("Can't close file %s\r\n", filename);
72         return -1;
73     }
74
75     return 0;
76 }
    
```

第 21 行, 数组 `usrdata` 是测试 APP 要向 `chrdevbase` 设备写入的数据。

第 35 行, 判断运行测试 APP 的时候输入的参数是不是为 3 个, `main` 函数的 `argc` 参数表示参数数量, `argv[]` 保存着具体的参数, 如果参数不为 3 个的话就表示测试 APP 用法错误。比如, 现在要从 `chrdevbase` 设备中读取数据, 需要输入如下命令:

```
./chrdevbaseApp /dev/chrdevbase 1
```

上述命令一共有三个参数“`./chrdevbaseApp`”、“`/dev/chrdevbase`”和“`1`”, 这三个参数分别对应 `argv[0]`、`argv[1]`和 `argv[2]`。第一个参数表示运行 `chrdevbaseAPP` 这个软件, 第二个参数表示测试 APP 要打开 `/dev/chrdevbase` 这个设备。第三个参数就是要执行的操作, 1 表示从 `chrdevbase`

中读取数据, 2 表示向 chrdevbase 写数据。

第 40 行, 获取要打开的设备文件名字, argv[1]保存着设备名字。

第 43 行, 调用 C 库中的 open 函数打开设备文件: /dev/chrdevbase。

第 49 行, 判断 argv[2]参数的值是 1 还是 2, 因为输入命令的时候其参数都是字符串格式的, 因此需要借助 atoi 函数将字符串格式的数字转换为真实的数字。

第 50 行, 当 argv[2]为 1 的时候表示要从 chrdevbase 设备中读取数据, 一共读取 50 字节的数据, 读取到的数据保存在 readbuf 中, 读取成功以后就在终端上打印出读取到的数据。

第 59 行, 当 argv[2]为 2 的时候表示要向 chrdevbase 设备写数据。

第 69 行, 对 chrdevbase 设备操作完成以后就关闭设备。

chrdevbaseApp.c 内容还是很简单的, 就是最普通的文件打开、关闭和读写操作。

5.4.3 编译驱动程序和测试 APP

1、编译驱动程序

首先编译驱动程序, 也就是 chrdevbase.c 这个文件, 我们需要将其编译为.ko 模块, 创建 Makefile 文件, 然后在其中输入如下内容:

示例代码 5.4.3.1 Makefile 文件

```

1 KERNELDIR := /home/alientek/rk3568_linux_sdk/kernel
2 CURRENT_PATH := $(shell pwd)
3 obj-m := chrdevbase.o
4
5 build: kernel_modules
6
7 kernel_modules:
8     $(MAKE) -C $(KERNELDIR) M=$(CURRENT_PATH) modules
9 clean:
10    $(MAKE) -C $(KERNELDIR) M=$(CURRENT_PATH) clean
    
```

第 1 行, KERNELDIR 表示开发板所使用的 Linux 内核源码目录, 使用绝对路径, 大家根据自己的实际情况填写即可。

第 2 行, CURRENT_PATH 表示当前路径, 直接通过运行“pwd”命令来获取当前所处路径。

第 3 行, obj-m 表示将 chrdevbase.c 这个文件编译为 chrdevbase.ko 模块。

第 8 行, 具体的编译命令, 后面的 modules 表示编译模块, -C 表示将当前的工作目录切换到指定目录中, 也就是 KERNELDIR 目录。M 表示模块源码目录, “make modules”命令中加入 M=dir 以后程序会自动到指定的 dir 目录中读取模块的源码并将其编译为.ko 文件。

Makefile 编写好以后输入命令编译驱动模块:

```
make ARCH=arm64 //ARCH=arm64 必须指定, 否则编译会失败
```

编译过程如图 5.4.3.1 所示:

```

alientek@ubuntu:~/Linux_Drivers/01_chrdevbase$ ls
01_chrdevbase.code-workspace chrdevbaseApp.c chrdevbase.c Makefile
alientek@ubuntu:~/Linux_Drivers/01_chrdevbase$ make ARCH=arm64
make -C /home/alientek/rk3568_linux_sdk/kernel M=/home/alientek/Linux_Drivers/01_chrdevbase modules
make[1]: 进入目录"/home/alientek/rk3568_linux_sdk/kernel"
  CC [M] /home/alientek/Linux_Drivers/01_chrdevbase/chrdevbase.o
Building modules, stage 2.
MODPOST 1 modules
  CC /home/alientek/Linux_Drivers/01_chrdevbase/chrdevbase.mod.o
  LD [M] /home/alientek/Linux_Drivers/01_chrdevbase/chrdevbase.ko
make[1]: 离开目录"/home/alientek/rk3568_linux_sdk/kernel"
alientek@ubuntu:~/Linux_Drivers/01_chrdevbase$
    
```

图 5.4.3.1 驱动模块编译过程

编译成功以后就会生成一个叫做 chrdevbaes.ko 的文件，此文件就是 chrdevbase 设备的驱动模块。至此，chrdevbase 设备的驱动就编译成功。

2、编译测试 APP

测试 APP 比较简单，只有一个文件，因此就不需要编写 Makefile 了，直接输入命令编译。因为测试 APP 是要在 ARM 开发板上运行的，所以需要使用我们前面安装好的交叉编译期来编译，输入如下命令：

```

/opt/atk-dlrk356x-toolchain/bin/aarch64-buildroot-linux-gnu-gcc chrdevbaseApp.c -o chrdevbaseApp
    
```

编译完成以后会生成一个叫做 chrdevbaseApp 的可执行程序，输入如下命令查看 chrdevbaseAPP 这个程序的文件信息：

```

file chrdevbaseApp
    
```

结果如图 5.4.3.2 所示：

```

alientek@ubuntu:~/Linux_Drivers/01_chrdevbase$ file chrdevbaseApp
chrdevbaseApp: ELF 64-bit LSB executable, ARM aarch64, version 1 (SYSV), dynamically linked, interp
reter /lib/ld-linux-aarch64.so.1, for GNU/Linux 3.7.0, not stripped
alientek@ubuntu:~/Linux_Drivers/01_chrdevbase$
    
```

图 5.4.3.2 chrdevbaseAPP 文件信息

从图 5.4.3.2 可以看出，chrdevbaseAPP 这个可执行文件是 32 位 LSB 格式，ARM 版本的，因此 chrdevbaseAPP 只能在 ARM 芯片下运行。

5.4.4 运行测试

1、使用 ADB 将驱动模块和测试 APP 发送到开发板

首先肯定是使用 adb 工具要将 Ubuntu 下编译得到的 chrdevbase.ko 和 chrdevbaseApp 这两个文件发送到开发板的/lib/modules/4.19.232 目录下，输入如下命令：

```

adb push chrdevbase.ko chrdevbaseApp /lib/modules/4.19.232
    
```

发送过程如图 5.4.4.1 所示：

```

alientek@ubuntu:~/Linux_Drivers/01_chrdevbase$ adb push chrdevbase.ko chrdevbaseApp /lib/modules/4.19.232
chrdevbase.ko: 1 file pushed. 1.1 MB/s (332104 bytes in 0.298s)
chrdevbaseApp: 1 file pushed. 0.5 MB/s (9464 bytes in 0.017s)
2 files pushed. 1.0 MB/s (341568 bytes in 0.326s)
alientek@ubuntu:~/Linux_Drivers/01_chrdevbase$
    
```

图 5.4.4.1 使用 adb 命令发送驱动和测试 APP 到开发板

从图 5.4.4.1 可以看出，这两个文件发送成功，此时开发板/lib/modules/4.19.232 目录下就有这来个文件了，如图 5.4.4.2 所示：

```

root@ATK-DLRK356X:/lib/modules/4.19.232# ls
chrdevbaseApp chrdevbase.ko
root@ATK-DLRK356X:/lib/modules/4.19.232#
    
```

图 5.4.4.2 开发板中的 chrdevbase.ko 和 chrdevbaseAPP 文件

2、加载驱动模块

输入如下命令加载 chrdevbase.ko 驱动文件:

```
modprobe chrdevbase.ko
```

或

```
insmod chrdevbase.ko
```

如果使用 modprobe 加载驱动的话, 可能会出现如图 5.4.4.3 或图 5.4.4.4 所示的提示:

```
root@ATK-DLRK356X:/lib/modules/4.19.232# modprobe chrdevbase.ko
modprobe: FATAL: Module chrdevbase.ko not found in directory /lib/modules/4.19.232
root@ATK-DLRK356X:/lib/modules/4.19.232#
```

图 5.4.4.3 找不到“chrdevbase.ko”文件

modprobe 命令会在“/lib/modules/4.19.232”目录下解析 modules.dep 文件, modules.dep 文件里面保存了要加载的.ko 模块, 我们不用手动创建 modules.dep 这个文件, 直接输入 depmod 命令即可自动生成 modules.dep, 有些根文件系统可能没有 depmod 这个命令, 如果没有这个命令就只能重新配置 busybox, 使能此命令, 然后重新编译 busybox。输入“depmod”命令以后会自动生成 modules.alias、modules.symbols 和 modules.dep 等等一些 modprobe 所需的文件, 如图 5.4.4.5 所示:

```
root@ATK-DLRK356X:/lib/modules/4.19.232# depmod
depmod: WARNING: could not open modules.order at /lib/modules/4.19.232: No such file or directory
depmod: WARNING: could not open modules.builtin at /lib/modules/4.19.232: No such file or directory
root@ATK-DLRK356X:/lib/modules/4.19.232# ls
chrdevbaseApp  modules.alias.bin  modules.dep.bin  modules.symbols
chrdevbase.ko  modules.builtin.bin  modules.devname  modules.symbols.bin
modules.alias  modules.dep        modules.softdep
root@ATK-DLRK356X:/lib/modules/4.19.232#
```

图 5.4.4.4 depmod 命令执行结果

重新使用 modprobe 加载 chrdevbase.ko, 结果如图 5.4.4.6 所示:

```
root@ATK-DLRK356X:/lib/modules/4.19.232# modprobe chrdevbase
[ 4406.577298] chrdevbase init!
root@ATK-DLRK356X:/lib/modules/4.19.232#
```

图 5.4.4.5 驱动加载成功

从图 5.4.4.6 可以看到“chrdevbase init!”这一行, 这一行正是 chrdevbase.c 中模块入口函数 chrdevbase_init 输出的信息, 说明模块加载成功!

注意: 用 modprobe 这个命令加载模块不用加“.ko”, 加了的话可能报会图 5.4.4.3 所示错误, 所以建议大家在使用 modprobe 命令加载驱动模块的时候不用加模块名字后面的“.ko”后缀。

输入“lsmod”命令即可查看当前系统中存在的模块, 结果如图 5.4.4.7 所示:

```
root@ATK-DLRK356X:/lib/modules/4.19.232# lsmod
Module                Size  Used by
chrdevbase            16384  0
8852bs                3895296  0
root@ATK-DLRK356X:/lib/modules/4.19.232#
```

chrdevbase 驱动模块

图 5.4.4.6 当前系统中的模块

从图 5.4.4.7 可以看出, 当前系统有三个驱动模块, 其中一个就是“chrdevbase”。输入如下命令查看当前系统中有没有 chrdevbase 这个设备:

```
cat /proc/devices
```

结果如图 5.4.4.8 所示:

```

root@ATK-DLRK356X:/lib/modules/4.19.232# cat /proc/devices
Character devices:
 1 mem
 4 /dev/vc/0
 4 tty
 4 ttyS
 5 /dev/tty
 5 /dev/console
 5 /dev/ptmx
 7 vcs
10 misc
13 input
29 fb
 81 video4linux
 89 i2c
 90 mtd
108 ppp
116 alsa
128 ptm
136 pts
153 spi
166 ttyACM
180 usb
188 ttyUSB
189 usb device
200 chrdevbase
216 rfcomm
226 drm
237 hidraw
238 rpmb
    
```

← 主设备号为chrdevbase设备

图 5.4.4.7 当前系统设备

从图 5.4.4.8 可以看出，当前系统存在 chrdevbase 这个设备，主设备号为 200，跟我们设置的主设备号一致。

3、创建设备节点文件

驱动加载成功需要在/dev 目录下创建一个与之对应的设备节点文件，应用程序就是通过操作这个设备节点文件来完成对具体设备的操作。输入如下命令创建/dev/chrdevbase 这个设备节点文件：

```
mknod /dev/chrdevbase c 200 0
```

其中“mknod”是创建节点命令，“/dev/chrdevbase”是要创建的节点文件，“c”表示这是个字符设备，“200”是设备的主设备号，“0”是设备的次设备号。创建完成以后就会存在 /dev/chrdevbase 这个文件，可以使用“ls /dev/chrdevbase -l”命令查看，结果如图 5.4.4.9 所示：

```

root@ATK-DLRK356X:/lib/modules/4.19.232# mknod /dev/chrdevbase c 200 0
root@ATK-DLRK356X:/lib/modules/4.19.232# ls /dev/chrdevbase -l
crw-r--r-- 1 root root 200, 0 May 26 11:45 /dev/chrdevbase
root@ATK-DLRK356X:/lib/modules/4.19.232#
    
```

图 5.4.4.8 /dev/chrdevbase 文件

如果 chrdevbaseAPP 想要读写 chrdevbase 设备，直接对/dev/chrdevbase 进行读写操作即可。相当于/dev/chrdevbase 这个文件是 chrdevbase 设备在用户空间中的实现。前面一直说 Linux 下一切皆文件，包括设备也是文件，现在大家应该是有这个概念了吧？

4、chrdevbase 设备操作测试

一切准备就绪，接下来就是“大考”的时刻了。使用 chrdevbaseApp 软件操作 chrdevbase 这个设备，看看读写是否正常，首先进行读操作，输入如下命令：

```
./chrdevbaseApp /dev/chrdevbase 1
```

结果如图 5.4.4.10 所示：

```

root@ATK-DLRK356X:/lib/modules/4.19.232# ./chrdevbaseApp /dev/chrdevbase 1
[ 6416.311625] kernel senddata ok!
read data:kernel data!
root@ATK-DLRK356X:/lib/modules/4.19.232#
    
```

图 5.4.4.9 读操作结果

从图 5.4.4.10 可以看出，首先输出“kernel senddata ok!”这一行信息，这是驱动程序中 chrdevbase_read 函数输出的信息，因为 chrdevbaseApp 使用 read 函数从 chrdevbase 设备读取数据，因此 chrdevbase_read 函数就会执行。chrdevbase_read 函数向 chrdevbaseApp 发送“kernel data!”数据，chrdevbaseApp 接收到以后就打印出来，“read data:kernel data!”就是 chrdevbaseApp 打印出来的接收到的数据。说明对 chrdevbase 的读操作正常，接下来测试对 chrdevbase 设备的写操作，输入如下命令：

```
./chrdevbaseApp /dev/chrdevbase 2
```

结果如图 5.4.4.11 所示：

```
root@ATK-DLRK356X:/lib/modules/4.19.232# ./chrdevbaseApp /dev/chrdevbase 2
[ 6556.120860] kernel recevdata:usr data!
root@ATK-DLRK356X:/lib/modules/4.19.232#
```

图 5.4.4.10 写操作结果

只有一行“kernel recevdata:usr data!”，这个是驱动程序中的 chrdevbase_write 函数输出的。chrdevbaseAPP 使用 write 函数向 chrdevbase 设备写入数据“usr data!”。chrdevbase_write 函数接收到以后将其打印出来。说明对 chrdevbase 的写操作正常，既然读写都没问题，说明我们编写的 chrdevbase 驱动是没有问题的。

5、卸载驱动模块

如果不再使用某个设备的话可以将其驱动卸载掉，比如输入如下命令卸载掉 chrdevbase 这个设备：

```
rmmod chrdevbase
```

卸载以后使用 lsmod 命令查看 chrdevbase 这个模块还存不存在，结果如图 5.4.4.12 所示：

```
root@ATK-DLRK356X:/lib/modules/4.19.232# rmmod chrdevbase
[ 6741.393386] chrdevbase exit!
root@ATK-DLRK356X:/lib/modules/4.19.232# lsmod
Module                Size  Used by
8852bs                 3895296  0
root@ATK-DLRK356X:/lib/modules/4.19.232#
```

图 5.4.4.11 系统中当前模块

从图 5.4.4.12 可以看出，此时系统已经没有 chrdevbase 这个模块了，说明模块卸载成功。

至此，chrdevbase 这个设备的整个驱动就验证完成了，驱动工作正常。本章我们详细的讲解了字符设备驱动的开发步骤，并且以一个虚拟的 chrdevbase 设备为例，带领大家完成了第一个字符设备驱动的开发，掌握了字符设备驱动的开发框架以及测试方法，以后的字符设备驱动实验基本都以此为蓝本。

第六章 嵌入式 Linux LED 驱动开发实验

上一章我们详细的讲解了字符设备驱动开发步骤，并且用一个虚拟的 `chrdevbase` 设备为例带领大家完成了第一个字符设备驱动的开发。本章我们就开始编写第一个真正的 Linux 字符设备驱动。在正点原子 ATK-DLRK3568 开发板上有一个 LED 灯，本章我们就来学习一下如何编写 Linux 下的 LED 灯驱动。

6.1 Linux 下 LED 灯驱动原理

Linux 下的任何外设驱动，最终都是要配置相应的硬件寄存器。所以本章的 LED 灯驱动最终也是对 RK3568 的 IO 口进行配置，与裸机实验不同的是，在 Linux 下编写驱动要符合 Linux 的驱动框架。开发板上的 LED 连接到 RK3568 的 GPIO0_C0 这个引脚上，因此本章实验的重点就是编写 Linux 下 RK3568 引脚控制驱动。

6.1.1 地址映射

在编写驱动之前，我们需要先简单了解一下 MMU 这个神器，MMU 全称叫做 Memory Manage Unit，也就是内存管理单元。在老版本的 Linux 中要求处理器必须有 MMU，但是现在 Linux 内核已经支持无 MMU 的处理器了。MMU 主要完成的功能如下：

- ①、完成虚拟空间到物理空间的映射。
- ②、内存保护，设置存储器的访问权限，设置虚拟存储空间的缓冲特性。

我们重点来看一下第①点，也就是虚拟空间到物理空间的映射，也叫做地址映射。首先了解两个地址概念：虚拟地址(VA,Virtual Address)、物理地址(PA, Physical Address)。对于 32 位的处理器来说，虚拟地址范围是 $2^{32}=4\text{GB}$ (64 位的处理器则是 $2^{64}=18.45 \times 10^{18} \text{GB}$ ，即从 0 到 $2^{64}-1$ 的范围。这个地址范围比 32 位处理器的地址范围要大得多，可以支持更大的内存空间，提高了计算机的性能)。例如我们的开发板上有 1GB 的 DDR3，这 1GB 的内存就是物理内存，经过 MMU 可以将其映射到整个 4GB 的虚拟空间，如图 6.1.2 所示：

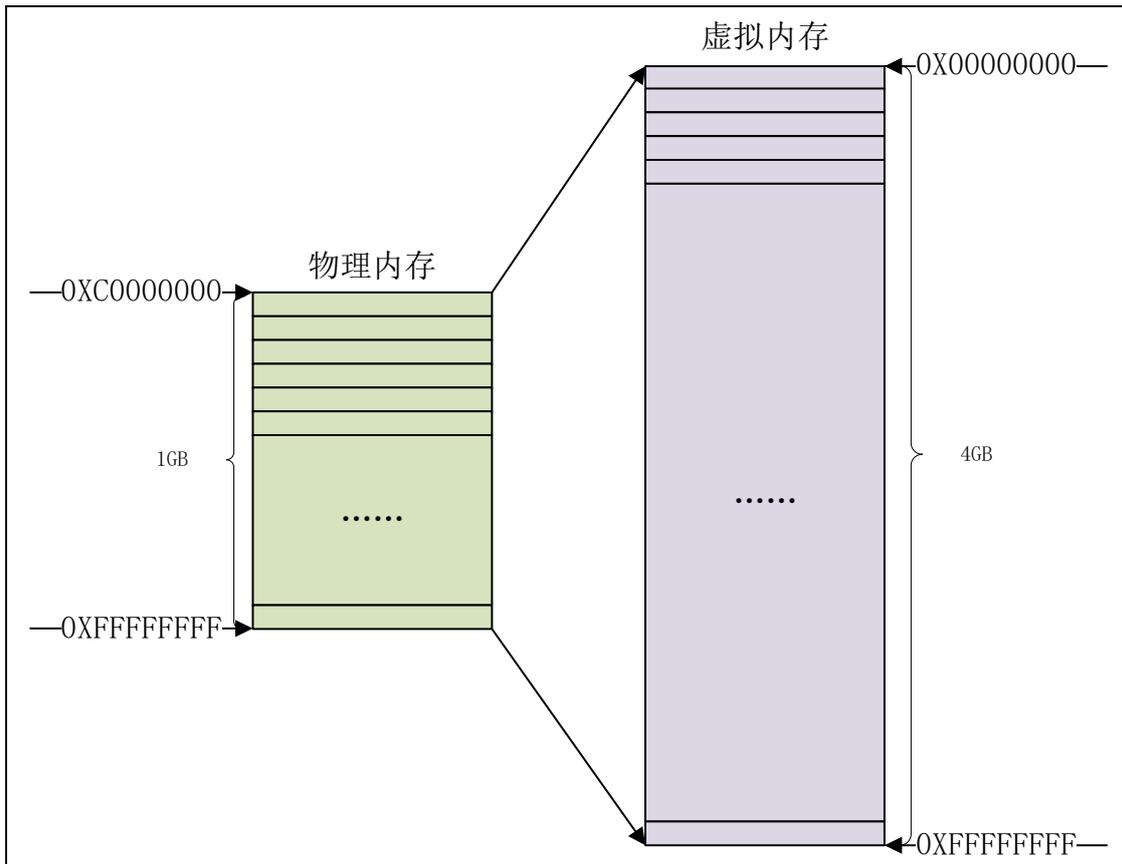


图 6.1.2 内存映射

物理内存只有 1GB, 虚拟内存有 4GB, 那么肯定存在多个虚拟地址映射到同一个物理地址上去, 虚拟地址范围比物理地址范围大的问题处理器自会处理, 这里我们不要去深究, 因为 MMU 是很复杂的一个东西。

Linux 内核启动的时候会初始化 MMU, 设置好内存映射, 设置好以后 CPU 访问的都是虚拟地址。比如 RK3568 的 GPIO0_C0 引脚的 IO 复用寄存器 PMU_GRP_GPIO0C_IOMUX_L 物理地址为 0xFDC20010。如果没有开启 MMU 的话直接向 0xFDC20010 这个寄存器地址写入数据就可以配置 GPIO0_C0 的引脚的复用功能。现在开启了 MMU, 并且设置了内存映射, 因此就不能直接向 0xFDC20010 这个地址写入数据了。我们必须得到 0xFDC20010 这个物理地址在 Linux 系统里面对应的虚拟地址, 这里就涉及到了物理内存和虚拟内存之间的转换, 需要用到两个函数: ioremap 和 iounmap。

1、ioremap 函数

ioremap 函数用于获取指定物理地址空间对应的虚拟地址空间, 定义在 arch/arm/include/asm/io.h 文件中, 定义如下:

示例代码 6.1.1.1 ioremap 函数声明

```
431 void __iomem *ioremap(resource_size_t res_cookie, size_t size);
```

函数的实现是在 arch/arm/mm/ioremap.c 文件中, 实现如下:

示例代码 6.1.1.2 ioremap 函数实现

```
376 void __iomem *ioremap(resource_size_t res_cookie, size_t size)
377 {
378     return arch_ioremap_caller(res_cookie, size, MT_DEVICE,
379                               __builtin_return_address(0));
380 }
381 EXPORT_SYMBOL(ioremap);
```

ioremap 有两个参数: res_cookie 和 size, 真正起作用的是函数 arch_ioremap_caller。ioremap 函数有两个参数和一个返回值, 这些参数和返回值的含义如下:

res_cookie: 要映射的物理起始地址。

size: 要映射的内存空间大小。

返回值: __iomem 类型的指针, 指向映射后的虚拟空间首地址。

假如我们要获取 RK3568 的 PMU_GRP_GPIO0C_IOMUX_L 寄存器对应的虚拟地址, 使用如下代码即可:

```
#define PMU_GRP_GPIO0C_IOMUX_L    (0xFDC20010)
static void __iomem*             PMU_GRP_GPIO0C_IOMUX_L_PI;
PMU_GRP_GPIO0C_IOMUX_L_PI = ioremap(PMU_GRP_GPIO0C_IOMUX_L, 4);
```

宏 PMU_GRP_GPIO0C_IOMUX_L 是寄存器物理地址, PMU_GRP_GPIO0C_IOMUX_L_PI 是映射后的虚拟地址。对于 RK3568 来说一个寄存器是 4 字节(32 位), 因此映射的内存长度为 4。映射完成以后直接对 PMU_GRP_GPIO0C_IOMUX_L_PI 进行读写操作即可。

2、iounmap 函数

卸载驱动的时候需要使用 iounmap 函数释放掉 ioremap 函数所做的映射, iounmap 函数原型如下:

示例代码 6.1.1.3 iounmap 函数原型

```
460 void iounmap (volatile void __iomem *addr)
```

`iounmap` 只有一个参数 `addr`, 此参数就是要取消映射的虚拟地址空间首地址。假如我们现在要取消掉 `PMU_GRF_GPIO0C_IOMUX_L_PI` 寄存器的地址映射, 使用如下代码即可:

```
iounmap(PMU_GRF_GPIO0C_IOMUX_L_PI);
```

6.1.2 I/O 内存访问函数

这里说的 I/O 是输入/输出的意思, 并不是我们学习单片机的时候讲的 GPIO 引脚。这里涉及到两个概念: I/O 端口和 I/O 内存。当外部寄存器或内存映射到 IO 空间时, 称为 I/O 端口。当外部寄存器或内存映射到内存空间时, 称为 I/O 内存。但是对于 ARM 来说没有 I/O 空间这个概念, 因此 ARM 体系下只有 I/O 内存(可以直接理解为内存)。使用 `ioremap` 函数将寄存器的物理地址映射到虚拟地址以后, 我们就可以直接通过指针访问这些地址, 但是 Linux 内核不建议这么做, 而是推荐使用一组操作函数来对映射后的内存进行读写操作。

1、读操作函数

读操作函数有如下几个:

示例代码 6.1.2.1 读操作函数

```
1 u8 readb(const volatile void __iomem *addr)
2 u16 readw(const volatile void __iomem *addr)
3 u32 readl(const volatile void __iomem *addr)
```

`readb`、`readw` 和 `readl` 这三个函数分别对应 8bit、16bit 和 32bit 读操作, 参数 `addr` 就是要读取写内存地址, 返回值就是读取到的数据。

2、写操作函数

写操作函数有如下几个:

示例代码 6.1.2.2 写操作函数

```
1 void writeb(u8 value, volatile void __iomem *addr)
2 void writew(u16 value, volatile void __iomem *addr)
3 void writel(u32 value, volatile void __iomem *addr)
```

`writeb`、`writew` 和 `writel` 这三个函数分别对应 8bit、16bit 和 32bit 写操作, 参数 `value` 是要写入的数值, `addr` 是要写入的地址。

6.2 硬件原理图分析

我们先进行 LED 的硬件原理分析, 打开开发板底板原理图, 底板原理图和核心板原理图都放到了开发板光盘中, 路径为: [开发板光盘](#) → [02、开发板原理图](#) → [01、底板原理图](#) → [ATK_DLRK3568 VX.X\(底板原理图\).pdf](#)。开发板上有一个 LED, 原理如图 6.2.1 所示:

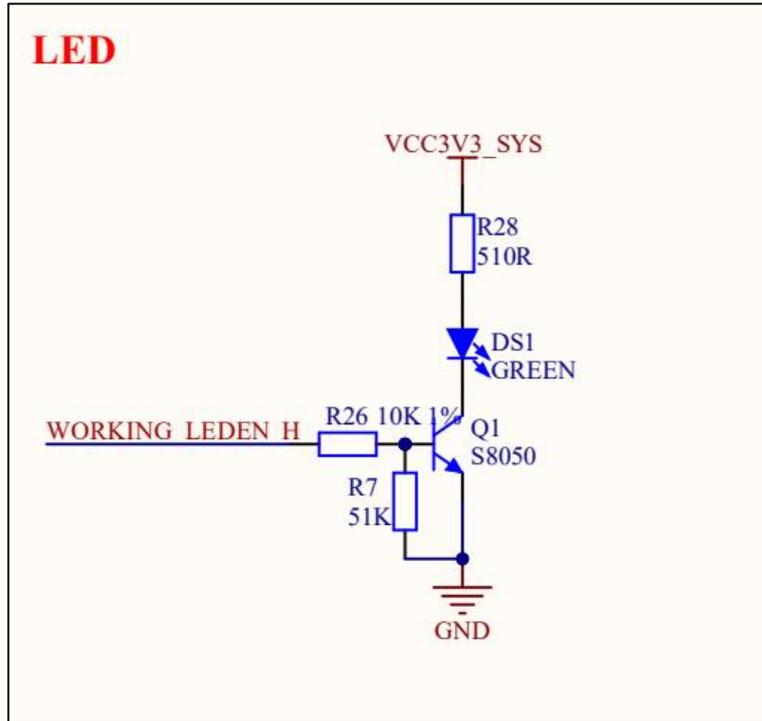


图 6.2.1 LED 原理图

从图 6.2.1 可以看出，LED 接到了 GPIO0_C0(WORKING_LEDN_H)上，当 GPIO0_C0 输出高电平(1)的时候 Q1 这个三极管就能导通，LED (DS1)这个绿色的发光二极管就会点亮。当 GPIO0_C0 输出低电平(0)的时候 Q1 这个三极管就会关闭，发光二极管 LED (DS1)不会导通，因此 LED 也就不会点亮。所以 LED 的亮灭取决于 GPIO0_C0 的输出电平，输出 1 就亮，输出 0 就灭。

6.3、RK3568 GPIO 驱动原理讲解

我们以 GPIO0_C0 为例，讲一下如何驱动 RK3568 的某一个 IO，应该做那些工作，操作哪些寄存器等。这里我们就要用到 RK3568 的参考手册，这个我们已经放到开发板资料里面了，路径：[开发板光盘](#)→03、核心板资料→核心板板载芯片资料→Rockchip RK3568 TRM Part1 V1.1-20210301.pdf (RK3568 参考手册 1) .pdf 和 [Rockchip RK3568 TRM Part2 V1.1-20210301 \(RK3568 参考手册 2\) .pdf](#)。

6.3.1 引脚复用设置

RK3568 的一个引脚一般用多个功能，也就是引脚复用，比如 GPIO0_C0 这个 IO 就可以用作：GPIO，PWM1_M0，GPU_AVS 和 UART0_RX 这四个功能，所以我们首先要设置好当前引脚用作什么功能，这里我们要使用 GPIO0_C0 的 GPIO 功能。

打开《[Rockchip RK3568 TRM Part1 V1.1-20210301 \(RK3568 参考手册 1\) .pdf](#)》这份文档，找到 PMU_GRP_GPIO0C_IOMUX_L 这个寄存器，寄存器描述如图 6.3.1.1 所示：

PMU GRF GPIO0C_IOMUX_L			
Address: Operational Base + offset (0x0010)			
Bit	Attr	Reset Value	Description
31:16	RW	0x0000	write_enable Write enable for lower 16bits, each bit is individual. 1'b0: Write access disable 1'b1: Write access enable
15	RO	0x0	reserved
14:12	RW	0x0	gpio0c3_sel 3'h0: GPIO0_C3 3'h1: PWM4 3'h2: VOP_PWMM0 3'h3: PCIE30X1_PERSTNM0 3'h4: MCU_JTAGTRSTN
11	RO	0x0	reserved
10:8	RW	0x0	gpio0c2_sel 3'h0: GPIO0_C2 3'h1: PWM3 3'h2: EDPDP_HPDM1 3'h3: PCIE30X1_WAKENM0 3'h4: MCU_JTAGTMS
7	RO	0x0	reserved
6:4	RW	0x0	gpio0c1_sel 3'h0: GPIO0_C1 3'h1: PWM2_M0 3'h2: NPU_AVS 3'h3: UART0_TX 3'h4: MCU_JTAGTDI
3	RO	0x0	reserved
2:0	RW	0x0	gpio0c0_sel 3'h0: GPIO0_C0 3'h1: PWM1_M0 3'h2: GPU_AVS 3'h3: UART0_RX

图 6.3.1.1 PMU_GRF_GPIO0C_IOMUX_L 寄存器描述

从图 6.3.1.1 可以看出 PMU_GRF_GPIO0C_IOMUX_L 寄存器地址为: base+offset, 其中 base 就是 PMU_GRF 外设的基地址, 为 0xFDC20000, offset 为 0x0010, 所以 PMU_GRF_GPIO0C_IOMUX_L 寄存器地址为 0xFDC20000+0x0010=0xFDC20010。

PMU_GRF_GPIO0C_IOMUX_L 寄存器分为 2 部分:

- ①、**bit31:16**: 低 16 位写使能位, 这 16 个 bit 控制着寄存器的低 16 位写使能。比如 bit16 就对应着 bit0 的写使能, 如要写 bit0, 那么 bit16 要置 1, 也就是允许对 bit0 进行写操作。
- ②、**bit15:0**: 功能设置位。

可以看出, PMU_GRF_GPIO0C_IOMUX_L 寄存器用于设置 GPIO0_C0~C3 这 4 个 IO 的复用功能, 其中 bit2:0 用于设置 GPIO0_C0 的复用功能, 有四个可选功能:

- 0**: GPIO0_C0
- 1**: PWM1_M0
- 2**: GPU_AVS
- 3**: UART0_RX

我们要将 GPIO0_C0 设置为 GPIO, 所以 PMU_GRF_GPIO0C_IOMUX_L 的 bit2:0 这三位设置 000。另外 bit18:16 要设置为 111, 允许写 bit2:0。

6.3.2 引脚驱动能力设置

RK3568 的 IO 引脚可以设置不同的驱动能力，GPIO0_C0 的驱动能力设置寄存器为 PMU_GRF_GPIO0C_DS_0，寄存器结构如图 6.3.2.1 所示：

PMU_GRF_GPIO0C_DS_0			
Address: Operational Base + offset (0x0090)			
Bit	Attr	Reset Value	Description
31:16	RW	0x0000	write_enable Write enable for lower 16bits, each bit is individual. 1'b0: Write access disable 1'b1: Write access enable
15:14	RO	0x0	reserved

Copyright 2021 © Rockchip Electronics Co., Ltd. 201

RKRK3568 TRM-Part1

Bit	Attr	Reset Value	Description
13:8	RW	0x01	gpio0c1_ds GPIO PAD Drive Strength control. 6'b000000: Disable 6'b000001: Level 0 6'b000011: Level 1 6'b000111: Level 2 6'b001111: Level 3 6'b011111: Level 4 6'b111111: Level 5 All other setting are reserved
7:6	RO	0x0	reserved
5:0	RW	0x01	gpio0c0_ds GPIO PAD Drive Strength control. 6'b000000: Disable 6'b000001: Level 0 6'b000011: Level 1 6'b000111: Level 2 6'b001111: Level 3 6'b011111: Level 4 6'b111111: Level 5 All other setting are reserved

图 6.3.2.1 PMU_GRF_GPIO0C_DS_0 寄存器

PMU_GRF_GPIO0C_DS_0 寄存器地址为: base+offset=0xFDC20000+0X0090=0xFDC20090。

PMU_GRF_GPIO0C_DS_0 寄存器也分为 2 部分：

- ①、bit31:16：低 16 位写使能位，这 16 个 bit 控制着寄存器的低 16 位写使能。比如 bit16 就对应着 bit15:0 的写使能，如要写 bit15:0，那么 bit16 要置 1，也就是允许对 bit15:0 进行写操作。

②、bit15:0: 功能设置位。

可以看出, PMU_GRF_GPIO0C_DS_0 寄存器用于设置 GPIO0_C0~C1 这 2 个 IO 的驱动能力, 其中 bit5:0 用于设置 GPIO0_C0 的驱动能力, 一共有 6 级。

这里我们将 GPIO0_C0 的驱动能力设置为 5 级, 所以 GRF_GPIO3D_DS_H 的 bit5:0 这六位设置 111111。另外 bit21:16 要设置为 111111, 允许写 bit5:0。

6.3.3 GPIO 输入输出设置

GPIO 是双向的, 也就是既可以做输入, 也可以做输出。本章我们使用 GPIO0_C0 来控制 LED 灯的亮灭, 因此要设置为输出。GPIO_SWPORT_DDR_L 和 GPIO_SWPORT_DDR_H 这两个寄存器用于设置 GPIO 的输入输出功能。RK3568 一共有 GPIO0、GPIO1、GPIO2、GPIO3 和 GPIO4 这五组 GPIO。其中 GPIO0~3 这四组每组都有 A0~A7、B0~B7、C0~C7 和 D0~D7 这 32 个 GPIO。每个 GPIO 需要一个 bit 来设置其输入输出功能, 一组 GPIO 就需要 32bit, GPIO_SWPORT_DDR_L 和 GPIO_SWPORT_DDR_H 这两个寄存器就是用来设置这一组 GPIO 所有引脚的输入输出功能的。其中 GPIO_SWPORT_DDR_L 设置的是低 16bit, GPIO_SWPORT_DDR_H 设置的是高 16bit。一组 GPIO 里面这 32 给引脚对应的 bit 如表 6.3.3.1 所示:

GPIO 组	GPIOX_A0~A7	GPIOX_B0~B7	GPIOX_C0~C7	GPIOX_D0~D7
对应的 bit	bit0~bit7	bit8~bit15	bit16~bit23	bit24~bit31

表 6.3.3.1 引脚对应的 bit

GPIO0_C0 很明显要用到 GPIO_SWPORT_DDR_H 寄存器, 寄存器描述如图 6.3.3.1 所示:

GPIO SWPORT DDR H			
Address: Operational Base + offset (0x000C)			
Bit	Attr	Reset Value	Description
31:16	WO	0x0000	write_mask Write enable for lower 16 bits, each bit is individual. 1'b0: Write access disable 1'b1: Write access enable
15:0	RW	0x0000	swport_ddr_high Data direction for the upper 16 bits of I/O Port, each bit is individual. 1'b0: Input 1'b1: Output Values written to this register independently control the direction of the corresponding data bit in the upper 16 bits of I/O Port.

图 6.3.3.1 GPIO_SWPORT_DDR_H 寄存器

GPIO_SWPORT_DDR_H 寄存器地址也是 base+offset, 其中 GPIO0~GPIO4 的基地址如表 6.3.3.2 所示:

GPIO 组	基地址
GPIO0	0xFDD60000
GPIO1	0xFE740000
GPIO2	0xFE750000
GPIO3	0xFE760000
GPIO4	0xFE770000

表 6.3.3.2 GPIO 基地址

所以 GPIO0_C0 对应的 GPIO_SWPORT_DDR_H 基地址就是 0xFDD60000+0X000C=0XFDD6000C。

GPIO_SWPORT_DDR_H 寄存器也分为 2 部分:

①、**bit31:16**: 低 16 位写使能位, 这 16 个 bit 控制着寄存器的低 16 位写使能。比如 bit16 就对应着 bit0 的写使能, 如要写 bit0, 那么 bit16 要置 1, 也就是允许对 bit0 进行写操作。

③、**bit15:0**: 功能设置位。

这里我们将 GPIO0_C0 设置为输出, 所以 GPIO_SWPORT_DDR_H 的 bit0 要置 1, 另外 bit16 要设置为 1, 允许写 bit16。

6.3.4 GPIO 引脚高低电平设置

GPIO 配置好以后就可以控制引脚输出高低电平了, 需要用到 [GPIO_SWPORT_DR_L](#) 和 [GPIO_SWPORT_DR_H](#) 这两个寄存器, 这两个原理和上面讲的 [GPIO_SWPORT_DDR_L](#) 和 [GPIO_SWPORT_DDR_H](#) 一样, 这里就不再赘述了。

GPIO0_C0 需要用到 GPIO_SWPORT_DR_H 寄存器, 寄存器描述如图 6.3.4.1 所示:

GPIO SWPORT DR H			
Address: Operational Base + offset (0x0004)			
Copyright 2021 © Rockchip Electronics Co., Ltd. 654			
RKRK3568 TRM-Part1			
Bit	Attr	Reset Value	Description
31:16	WO	0x0000	write_mask Write enable for lower 16 bits, each bit is individual. 1'b0: Write access disable 1'b1: Write access enable
15:0	RW	0x0000	swport_dr_high Output data for the upper 16 bits of I/O Port, each bit is individual. 1'b0: Low 1'b1: High Values written to this register are output on the I/O signals for the upper 16 bits of I/O Port if the corresponding data direction bits for I/O Port are set to Output mode. The value read back is equal to the last value written to this register.

图 6.3.4.1 [GPIO_SWPORT_DR_H](#) 寄存器

同样的, GPIO0_C0 对应 bit0, 如果要输出低电平, 那么 bit0 置 0, 如果要输出高电平, bit0 置 1。bit16 也要置 1, 允许写 bit0。

关于 RK3568 的 GPIO 配置原理就讲到这里。

6.4 实验程序编写

本实验对应的例程路径为: [开发板光盘](#) → [06、Linux 驱动例程源码](#) → [02_led](#)。

本章实验编写 Linux 下的 LED 灯驱动, 可以通过应用程序对开发板上的 LED0 进行开关操作。

6.4.1 LED 灯驱动程序编写

新建名为“02_led”文件夹，然后在 02_led 文件夹里面创建 VSCode 工程，工作区命名为“led”。工程创建好以后新建 led.c 文件，此文件就是 led 的驱动文件，在 led.c 里面输入如下内容：

示例代码 6.4.1.1 led.c 驱动文件代码

```

1  #include <linux/types.h>
2  #include <linux/kernel.h>
3  #include <linux/delay.h>
4  #include <linux/ide.h>
5  #include <linux/init.h>
6  #include <linux/module.h>
7  #include <linux/errno.h>
8  #include <linux/gpio.h>
9  // #include <asm/mach/map.h>
10 #include <asm/uaccess.h>
11 #include <asm/io.h>
12 /*****
13 Copyright © ALIENTEK Co., Ltd. 1998-2029. All rights reserved.
14 文件名      : led.c
15 作者        : 正点原子
16 版本        : V1.0
17 描述        : LED 驱动文件。
18 其他        : 无
19 论坛        : www.openedv.com
20 日志        : 初版 v1.0 2022/12/02 正点原子团队创建
21 *****/
22 #define LED_MAJOR      200    /* 主设备号 */
23 #define LED_NAME      "led"  /* 设备名字 */
24
25 #define LEDOFF  0          /* 关灯 */
26 #define LEDON   1          /* 开灯 */
27
28 #define PMU_GRP_BASE      (0xFDC20000)
29 #define PMU_GRP_GPIO0C_IOMUX_L      (PMU_GRP_BASE + 0x0010)
30 #define PMU_GRP_GPIO0C_DS_0      (PMU_GRP_BASE + 0x100EC)
31
32 #define GPIO0_BASE      (0xFDD60000)
33 #define GPIO0_SWPORT_DR_H      (GPIO0_BASE + 0x0004)
34 #define GPIO0_SWPORT_DDR_H      (GPIO0_BASE + 0x000C)
35
36 /* 映射后的寄存器虚拟地址指针 */
37 static void __iomem *PMU_GRP_GPIO0C_IOMUX_L_PI;
    
```

```

38 static void __iomem *PMU_GRF_GPIO0C_DS_0_PI;
39 static void __iomem *GPIO0_SWPORT_DR_H_PI;
40 static void __iomem *GPIO0_SWPORT_DDR_H_PI;
41
42 /*
43  * @description      : LED 打开/关闭
44  * @param - sta      : LEDON(0) 打开 LED, LEDOFF(1) 关闭 LED
45  * @return           : 无
46  */
47 void led_switch(u8 sta)
48 {
49     u32 val = 0;
50     if(sta == LEDON) {
51         val = readl(GPIO0_SWPORT_DR_H_PI);
52         val &= ~(0X1 << 0); /* bit0 清零*/
53         val |= ((0X1 << 16) | (0X1 << 0)); /* bit16 置 1, 允许写 bit0,
54                                             bit0, 高电平*/
55         writel(val, GPIO0_SWPORT_DR_H_PI);
56     }else if(sta == LEDOFF) {
57         val = readl(GPIO0_SWPORT_DR_H_PI);
58         val &= ~(0X1 << 0); /* bit0 清零*/
59         val |= ((0X1 << 16) | (0X0 << 0)); /* bit16 置 1, 允许写 bit0,
60                                             bit0, 低电平 */
61         writel(val, GPIO0_SWPORT_DR_H_PI);
62     }
63 }
64
65 /*
66  * @description      : 物理地址映射
67  * @return           : 无
68  */
69 void led_remap(void)
70 {
71     PMU_GRF_GPIO0C_IOMUX_L_PI = ioremap(PMU_GRF_GPIO0C_IOMUX_L, 4);
72     PMU_GRF_GPIO0C_DS_0_PI = ioremap(PMU_GRF_GPIO0C_DS_0, 4);
73     GPIO0_SWPORT_DR_H_PI = ioremap(GPIO0_SWPORT_DR_H, 4);
74     GPIO0_SWPORT_DDR_H_PI = ioremap(GPIO0_SWPORT_DDR_H, 4);
75 }
76
77 /*
78  * @description      : 取消映射
79  * @return           : 无
80  */
    
```

```

81 void led_unmap(void)
82 {
83     /* 取消映射 */
84     iounmap(PMU_GRF_GPIO0C_IOMUX_L_PI);
85     iounmap(PMU_GRF_GPIO0C_DS_0_PI);
86     iounmap(GPIO0_SWPORT_DR_H_PI);
87     iounmap(GPIO0_SWPORT_DDR_H_PI);
88 }
89
90 /*
91  * @description      : 打开设备
92  * @param - inode    : 传递给驱动的 inode
93  * @param - filp     : 设备文件, file 结构体有个叫做 private_data 的成员变量
94  *                   一般在 open 的时候将 private_data 指向设备结构体。
95  * @return           : 0 成功;其他 失败
96  */
97 static int led_open(struct inode *inode, struct file *filp)
98 {
99     return 0;
100 }
101
102 /*
103  * @description      : 从设备读取数据
104  * @param - filp     : 要打开的设备文件(文件描述符)
105  * @param - buf      : 返回给用户空间的数据缓冲区
106  * @param - cnt      : 要读取的数据长度
107  * @param - offt     : 相对于文件首地址的偏移
108  * @return           : 读取的字节数, 如果为负值, 表示读取失败
109  */
110 static ssize_t led_read(struct file *filp, char __user *buf, size_t
cnt, loff_t *offt)
111 {
112     return 0;
113 }
114
115 /*
116  * @description      : 向设备写数据
117  * @param - filp     : 设备文件, 表示打开的文件描述符
118  * @param - buf      : 要写给设备写入的数据
119  * @param - cnt      : 要写入的数据长度
120  * @param - offt     : 相对于文件首地址的偏移
121  * @return           : 写入的字节数, 如果为负值, 表示写入失败
122  */

```

```
123 static ssize_t led_write(struct file *filp, const char __user *buf,
124 size_t cnt, loff_t *offt)
125 {
126     int retvalue;
127     unsigned char databuf[1];
128     unsigned char ledstat;
129
130     retvalue = copy_from_user(databuf, buf, cnt);
131     if(retvalue < 0) {
132         printk("kernel write failed!\r\n");
133         return -EFAULT;
134     }
135     ledstat = databuf[0];    /* 获取状态值 */
136
137     if(ledstat == LEDON) {
138         led_switch(LEDON);    /* 打开 LED 灯 */
139     } else if(ledstat == LEDOFF) {
140         led_switch(LEDOFF);    /* 关闭 LED 灯 */
141     }
142     return 0;
143 }
144
145 /*
146 * @description      : 关闭/释放设备
147 * @param - filp     : 要关闭的设备文件(文件描述符)
148 * @return           : 0 成功;其他 失败
149 */
150 static int led_release(struct inode *inode, struct file *filp)
151 {
152     return 0;
153 }
154
155 /* 设备操作函数 */
156 static struct file_operations led_fops = {
157     .owner = THIS_MODULE,
158     .open = led_open,
159     .read = led_read,
160     .write = led_write,
161     .release = led_release,
162 };
163
164 /*
```

```

165 * @description : 驱动出口函数
166 * @param      : 无
167 * @return     : 无
168 */
169 static int __init led_init(void)
170 {
171     int retvalue = 0;
172     u32 val = 0;
173
174     /* 初始化 LED */
175     /* 1、寄存器地址映射 */
176     led_remap();
177
178     /* 2、设置 GPIO0_C0 为 GPIO 功能。*/
179     val = readl(PMU_GRF_GPIO0C_IOMUX_L_PI);
180     val &= ~(0X7 << 0); /* bit2:0, 清零 */
181     val |= ((0X7 << 16) | (0X0 << 0)); /* bit18:16 置 1, 允许写
bit2:0,
182                                     bit2:0: 0, 用作 GPIO0_C0 */
183     writel(val, PMU_GRF_GPIO0C_IOMUX_L_PI);
184
185     /* 3、设置 GPIO0_C0 驱动能力为 level5 */
186     val = readl(PMU_GRF_GPIO0C_DS_0_PI);
187     val &= ~(0X3F << 0); /* bit5:0 清零*/
188     val |= ((0X3F << 16) | (0X3F << 0)); /* bit21:16 置 1, 允许写
bit5:0,
189                                     bit5:0: 0, 用作 GPIO0_C0 */
190     writel(val, PMU_GRF_GPIO0C_DS_0_PI);
191
192     /* 4、设置 GPIO0_C0 为输出 */
193     val = readl(GPIO0_SWPORT_DDR_H_PI);
194     val &= ~(0X1 << 0); /* bit0 清零*/
195     val |= ((0X1 << 16) | (0X1 << 0)); /* bit16 置 1, 允许写 bit0,
196                                     bit0, 高电平 */
197     writel(val, GPIO0_SWPORT_DDR_H_PI);
198
199     /* 5、设置 GPIO0_C0 为低电平, 关闭 LED 灯。*/
200     val = readl(GPIO0_SWPORT_DR_H_PI);
201     val &= ~(0X1 << 0); /* bit0 清零*/
202     val |= ((0X1 << 16) | (0X0 << 0)); /* bit16 置 1, 允许写 bit0,
203                                     bit0, 低电平 */
204     writel(val, GPIO0_SWPORT_DR_H_PI);
205

```

```

206     /* 6、注册字符设备驱动 */
207     retvalue = register_chrdev(LED_MAJOR, LED_NAME, &led_fops);
208     if(retvalue < 0) {
209         printk("register chrdev failed!\r\n");
210         goto fail_map;
211     }
212     return 0;
213
214 fail_map:
215     led_unmap();
216     return -EIO;
217 }
218
219 /*
220 * @description : 驱动出口函数
221 * @param       : 无
222 * @return      : 无
223 */
224 static void __exit led_exit(void)
225 {
226     /* 取消映射 */
227     led_unmap();
228
229     /* 注销字符设备驱动 */
230     unregister_chrdev(LED_MAJOR, LED_NAME);
231 }
232
233 module_init(led_init);
234 module_exit(led_exit);
235 MODULE_LICENSE("GPL");
236 MODULE_AUTHOR("ALIENTEK");
237 MODULE_INFO(intree, "Y");
    
```

第 22~26 行，定义了一些宏，包括主设备号、设备名字、LED 开/关宏。

第 28~34 行，本实验要用到的寄存器宏定义。

第 37~40 行，经过内存映射以后的寄存器地址指针。

第 47~63 行，led_switch 函数，用于控制开发板上的 LED 灯亮灭，当参数 sta 为 LEDON(0) 的时候打开 LED 灯，sta 为 LEDOFF(1) 的时候关闭 LED 灯。

第 69~75 行，led_remap 函数，通过 ioremap 函数获取物理寄存器地址映射后的虚拟地址。

第 81~88 行，led_unmap 函数，取消所有物理寄存器映射，回收对应的资源。当程序出错退出或者卸载驱动模块的时候需要调用此函数，用来取消此前所做的寄存器映射。

第 97~100 行，led_open 函数，为空函数，可以自行在此函数中添加相关内容，一般在此函数中将设备结构体作为参数 filp 的私有数据(filp->private_data)，后面实验会讲解如何添加私有数据。

第 110~113 行, `led_read` 函数, 为空函数, 如果想在应用程序中读取 LED 的状态, 那么就可以在此函数中添加相应的代码。

第 123~143 行, `led_write` 函数, 实现对 LED 灯的开关操作, 当应用程序调用 `write` 函数向 `led` 设备写数据的时候此函数就会执行。首先通过函数 `copy_from_user` 获取应用程序发送过来的操作信息(打开还是关闭 LED), 最后根据应用程序的操作信息来打开或关闭 LED 灯。

第 150~153 行, `led_release` 函数, 为空函数, 可以自行在此函数中添加相关内容, 一般关闭设备的时候会释放掉 `led_open` 函数中添加的私有数据。

第 156~162 行, 设备文件操作结构体 `led_fops` 的定义和初始化。

第 169~217 行, 驱动入口函数 `led_init`, 此函数实现了 LED 的初始化工作,。比如设置 `GPIO0_D4` 的复用功能、设置驱动能力等级、配置输出功能、设置默认电平等等。最后, 最重要的一步! 使用 `register_chrdev` 函数注册 `led` 这个字符设备。

第 214~216 行, 如果前面注册字符设备失败, 就要回收以前注册成功的资源。

第 224~231 行, 驱动出口函数 `led_exit`, 首先使用函数 `iounmap` 取消内存映射, 最后使用函数 `unregister_chrdev` 注销 `led` 这个字符设备。

第 233~234 行, 使用 `module_init` 和 `module_exit` 这两个函数指定 `led` 设备驱动加载和卸载函数。

第 235~236 行, 添加 LICENSE 和作者信息。

第 237 行, 告诉内核这个驱动也是 `intree` 模块驱动。

6.4.2 编写测试 APP

编写测试 APP, `led` 驱动加载成功以后手动创建 `/dev/led` 节点, 应用程序(APP)通过操作 `/dev/led` 文件来完成对 LED 设备的控制。向 `/dev/led` 文件写 0 表示关闭 LED 灯, 写 1 表示打开 LED 灯。新建 `ledApp.c` 文件, 在里面输入如下内容:

示例代码 6.4.2.1 `ledApp.c` 文件代码

```

1  #include "stdio.h"
2  #include "unistd.h"
3  #include "sys/types.h"
4  #include "sys/stat.h"
5  #include "fcntl.h"
6  #include "stdlib.h"
7  #include "string.h"
8  /*****
9  Copyright © ALIENTEK Co., Ltd. 1998-2029. All rights reserved.
10  文件名      : ledApp.c
11  作者        : 正点原子
12  版本        : V1.0
13  描述        : led 测试 APP。
14  其他        : 无
15  使用方法    : ./ledtest /dev/led 0 关闭 LED
16              : ./ledtest /dev/led 1 打开 LED
17  论坛        : www.openedv.com
18  日志        : 初版 V1.0 2022/12/02 正点原子团队创建
19  *****/
    
```

```

20
21 #define LEDOFF  0
22 #define LEDON   1
23
24 /*
25  * @description      : main 主程序
26  * @param - argc     : argv 数组元素个数
27  * @param - argv     : 具体参数
28  * @return           : 0 成功;其他 失败
29  */
30 int main(int argc, char *argv[])
31 {
32     int fd, retvalue;
33     char *filename;
34     unsigned char databuf[1];
35
36     if(argc != 3){
37         printf("Error Usage!\r\n");
38         return -1;
39     }
40
41     filename = argv[1];
42
43     /* 打开 led 驱动 */
44     fd = open(filename, O_RDWR);
45     if(fd < 0){
46         printf("file %s open failed!\r\n", argv[1]);
47         return -1;
48     }
49
50     databuf[0] = atoi(argv[2]); /* 要执行的操作: 打开或关闭 */
51
52     /* 向/dev/led 文件写入数据 */
53     retvalue = write(fd, databuf, sizeof(databuf));
54     if(retvalue < 0){
55         printf("LED Control Failed!\r\n");
56         close(fd);
57         return -1;
58     }
59
60     retvalue = close(fd); /* 关闭文件 */
61     if(retvalue < 0){
62         printf("file %s close failed!\r\n", argv[1]);

```

```

63     return -1;
64 }
65     return 0;
66 }
    
```

ledApp.c 的内容还是很简单的，就是对 led 的驱动文件进行最基本的打开、关闭、写操作等。

6.5 运行测试

6.5.1 编译驱动程序和测试 APP

1、编译驱动程序

编写 Makefile 文件，本章实验的 Makefile 文件和第五章实验基本一样，只是将 obj-m 变量的值改为 led.o，Makefile 内容如下所示：

示例代码 6.5.1.1 Makefile 文件

```

1  KERNELDIR := /home/alientek/rk3568_linux_sdk/kernel
.....
4  obj-m := led.o
.....
11 clean:
12 $(MAKE) -C $(KERNELDIR) M=$(CURRENT_PATH) clean
    
```

第 4 行，设置 obj-m 变量的值为 led.o。

输入如下命令编译出驱动模块文件：

```
make ARCH=arm64 //ARCH=arm64 必须指定，否则编译会失败
```

编译成功以后就会生成一个名为“led.ko”的驱动模块文件。

2、编译测试 APP

输入如下命令编译测试 ledApp.c 这个测试程序：

```
/opt/atk-dlrk356x-toolchain/bin/aarch64-buildroot-linux-gnu-gcc ledApp.c -o ledApp
```

编译成功以后就会生成 ledApp 这个应用程序。

6.5.2 运行测试

1、关闭心跳灯

正点原子出厂系统将 LED 这个绿色的 LED 灯设置成了系统心跳灯，大家应该能看到这个板子上这个红色的 LED 灯一闪一闪的，提示系统正在运行。很明显，这个会干扰我们本章实验结果，需要先临时关闭系统心跳灯功能，在开发板中输入如下命令：

```
echo none > /sys/class/leds/work/trigger
```

上述命令就是临时关闭 LED 的心跳灯功能，开发板重启以后 LED 又会重新作为心跳灯。要想永久关闭 LED0 的心跳灯功能，需要修改设备树，这个我们后面会讲怎么将一个 LED 灯用作心跳灯。

2、加载并测试驱动

在 Ubuntu 中将上一小节编译出来的 led.ko 和 ledApp 这两个文件通过 adb 命令发送到开发板的/lib/modules/4.19.232 目录下，命令如下：

```
adb push led.ko ledApp /lib/modules/4.19.232
```

发送成功以后进入到目录 `lib/modules/4.19.232` 中, 输入如下命令加载 `led.ko` 驱动模块:

```
depmod //第一次加载驱动的时候需要运行此命令
```

```
modprobe led //加载驱动
```

驱动加载成功以后创建 “`/dev/led`” 设备节点, 命令如下:

```
mknod /dev/led c 200 0
```

驱动节点创建成功以后就可以使用 `ledApp` 软件来测试驱动是否工作正常, 输入如下命令打开 LED 灯:

```
./ledApp /dev/led 1 //打开 LED 灯
```

输入上述命令以后观察开发板上的绿色 LED 灯, 也就是 LED0 是否点亮, 如果点亮的话说明驱动工作正常。在输入如下命令关闭 LED 灯:

```
./ledApp /dev/led 0 //关闭 LED 灯
```

输入上述命令以后观察开发板上的绿色 LED 灯是否熄灭, 如果熄灭的话说明我们编写的 LED 驱动工作完全正常! 至此, 我们成功编写了第一个真正的 Linux 驱动设备程序。

如果要卸载驱动的话输入如下命令即可:

```
rmmod led
```

第七章 新字符设备驱动实验

经过前两章实验的实战操作, 我们已经掌握了 Linux 字符设备驱动开发的基本步骤, 字符设备驱动开发重点是使用 `register_chrdev` 函数注册字符设备, 当不再使用设备的时候就使用 `unregister_chrdev` 函数注销字符设备, 驱动模块加载成功以后还需要手动使用 `mknod` 命令创建设备节点。 `register_chrdev` 和 `unregister_chrdev` 这两个函数是老版本驱动使用的函数, 现在新的字符设备驱动已经不再使用这两个函数, 而是使用 Linux 内核推荐的新字符设备驱动 API 函数。本节我们就来学习一下如何编写新字符设备驱动, 并且在驱动模块加载的时候自动创建设备节点文件。

7.1 新字符设备驱动原理

7.1.1 分配和释放设备号

使用 `register_chrdev` 函数注册字符设备的时候只需要给定一个主设备号即可，但是这样会带来两个问题：

①、需要我们事先确定好哪些主设备号没有使用。

②、会将一个主设备号下的所有次设备号都使用掉，比如现在设置 LED 这个主设备号为 200，那么 0~1048575($2^{20}-1$)这个区间的次设备号就全部都被 LED 一个设备分走了。这样太浪费次设备号了！一个 LED 设备肯定只能有一个主设备号，一个次设备号。

解决这两个问题最好的方法就是在使用设备号的时候向 Linux 内核申请，需要几个就申请几个，由 Linux 内核分配设备可以使用的设备号。这个就是我们在 5.3.2 小节讲解的设备号的分配，如果没有指定设备号的话就使用如下函数来申请设备号：

```
int alloc_chrdev_region(dev_t *dev, unsigned baseminor, unsigned count, const char *name)
```

如果给定了设备的主设备号和次设备号就使用如下所示函数来注册设备号即可：

```
int register_chrdev_region(dev_t from, unsigned count, const char *name)
```

参数 `from` 是要申请的起始设备号，也就是给定的设备号；参数 `count` 是要申请的数量，一般都是一个；参数 `name` 是设备名字。

注销字符设备之后要释放掉设备号，不管是通过 `alloc_chrdev_region` 函数还是 `register_chrdev_region` 函数申请的设备号，统一使用如下释放函数：

```
void unregister_chrdev_region(dev_t from, unsigned count)
```

新字符设备驱动下，设备号分配示例代码如下：

示例代码 7.1.1.1 新字符设备驱动下设备号分配

```
1 int major; /* 主设备号 */
2 int minor; /* 次设备号 */
3 dev_t devid; /* 设备号 */
4
5 if (major) { /* 定义了主设备号 */
6     devid = MKDEV(major, 0); /* 大部分驱动次设备号都选择 0 */
7     register_chrdev_region(devid, 1, "test");
8 } else { /* 没有定义设备号 */
9     alloc_chrdev_region(&devid, 0, 1, "test"); /* 申请设备号 */
10    major = MAJOR(devid); /* 获取分配号的主设备号 */
11    minor = MINOR(devid); /* 获取分配号的次设备号 */
12 }
```

第 1~3 行，定义了主/次设备号变量 `major` 和 `minor`，以及设备号变量 `devid`。

第 5 行，判断主设备号 `major` 是否有效，在 Linux 驱动中一般给出主设备号的话就表示这个设备的设备号已经确定了，因为次设备号基本上都选择 0，这个算 Linux 驱动开发中约定俗成的一种规定了。

第 6 行，如果 `major` 有效的话就使用 `MKDEV` 来构建设备号，次设备号选择 0。

第 7 行，使用 `register_chrdev_region` 函数来注册设备号。

第 9~11 行，如果 `major` 无效，那就表示没有给定设备号。此时就要使用 `alloc_chrdev_region` 函数来申请设备号。设备号申请成功以后使用 `MAJOR` 和 `MINOR` 来提取出主设备号和次设备号，当然了，第 10 和 11 行提取主设备号和次设备号的代码可以不要。

如果要注销设备号的话, 使用如下代码即可:

示例代码 7.1.1.2 cdev 结构体

```
1 unregister_chrdev_region(devid, 1);          /* 注销设备号 */
```

注销设备号的代码很简单。

7.1.2 新的字符设备注册方法

1、字符设备结构

在 Linux 中使用 cdev 结构体表示一个字符设备, cdev 结构体在 include/linux/cdev.h 文件中的定义如下:

示例代码 7.1.2.1 cdev 结构体

```
1 struct cdev {
2     struct kobject    kobj;
3     struct module     *owner;
4     const struct file_operations *ops;
5     struct list_head list;
6     dev_t             dev;
7     unsigned int      count;
8 } __randomize_layout;
```

在 cdev 中有两个重要的成员变量: ops 和 dev, 这两个就是字符设备文件操作函数集合 file_operations 以及设备号 dev_t。编写字符设备驱动之前需要定义一个 cdev 结构体变量, 这个变量就表示一个字符设备, 如下所示:

```
struct cdev test_cdev;
```

2、cdev_init 函数

定义好 cdev 变量以后就要使用 cdev_init 函数对其进行初始化, cdev_init 函数原型如下:

```
void cdev_init(struct cdev *cdev, const struct file_operations *fops)
```

参数 cdev 就是要初始化的 cdev 结构体变量, 参数 fops 就是字符设备文件操作函数集合。

使用 cdev_init 函数初始化 cdev 变量的示例代码如下:

示例代码 7.1.2.2 cdev_init 函数使用示例代码

```
1 struct cdev testcdev;
2
3 /* 设备操作函数 */
4 static struct file_operations test_fops = {
5     .owner = THIS_MODULE,
6     /* 其他具体的初始项 */
7 };
8
9 testcdev.owner = THIS_MODULE;
10 cdev_init(&testcdev, &test_fops); /* 初始化 cdev 结构体变量 */
```

3、cdev_add 函数

cdev_add 函数用于向 Linux 系统添加字符设备(cdev 结构体变量), 首先使用 cdev_init 函数完成对 cdev 结构体变量的初始化, 然后使用 cdev_add 函数向 Linux 系统添加这个字符设备。

cdev_add 函数原型如下:

```
int cdev_add(struct cdev *p, dev_t dev, unsigned count)
```

参数 p 指向要添加的字符设备(cdev 结构体变量), 参数 dev 就是设备所使用的设备号, 参数 count 是要添加的设备数量。完善示例代码 7.1.2.2, 加入 cdev_add 函数, 内容如下所示:

示例代码 7.1.2.2 cdev_add 函数使用示例

```
1 struct cdev testcdev;
2
3 /* 设备操作函数 */
4 static struct file_operations test_fops = {
5     .owner = THIS_MODULE,
6     /* 其他具体的初始项 */
7 };
8
9 testcdev.owner = THIS_MODULE;
10 cdev_init(&testcdev, &test_fops);      /* 初始化 cdev 结构体变量 */
11 cdev_add(&testcdev, devid, 1);        /* 添加字符设备 */
```

示例代码 7.1.2.2 就是新的注册字符设备代码段, Linux 内核中大量的字符设备驱动都是采用这种方法向 Linux 内核添加字符设备。如果在加上示例代码 7.1.1.1 中分配设备号的程序, 那么它们就一起实现的就是函数 register_chrdev 的功能。

3、cdev_del 函数

卸载驱动的时候一定要使用 cdev_del 函数从 Linux 内核中删除相应的字符设备, cdev_del 函数原型如下:

```
void cdev_del(struct cdev *p)
```

参数 p 就是要删除的字符设备。如果要删除字符设备, 参考如下代码:

示例代码 7.1.2.3 cdev_del 函数使用示例

```
1 cdev_del(&testcdev); /* 删除 cdev */
```

cdev_del 和 unregister_chrdev_region 这两个函数合起来的功能相当于 unregister_chrdev 函数。

7.2 自动创建设备节点

在前面的 Linux 驱动实验中, 当我们使用 modprobe 加载驱动程序以后还需要使用命令 “mknod” 手动创建设备节点。本节就来讲解一下如何实现自动创建设备节点, 在驱动中实现自动创建设备节点的功能以后, 使用 modprobe 加载驱动模块成功的话就会自动在/dev 目录下创建对应的设备文件。

7.2.1 mdev 机制

udev 是一个用户程序, 在 Linux 下通过 udev 来实现设备文件的创建与删除, udev 可以检测系统中硬件设备状态, 可以根据系统中硬件设备状态来创建或者删除设备文件。比如使用 modprobe 命令成功加载驱动模块以后就自动在/dev 目录下创建对应的设备节点文件, 使用 rmmod 命令卸载驱动模块以后就删除掉/dev 目录下的设备节点文件。

开发板启动的时候会启动 udev, 如图 7.2.1.1 所示:

```
Starting syslogd: OK
Starting klogd: OK
Running sysctl: OK
/usr/bin/modetest
populating /dev using udev: [ 3.831623] udevd[163]: starting version 3.2.7
[ 3.838860] udevd[163]: specified group 'kvm' unknown
[ 3.850123] udevd[164]: starting eudev-3.2.7
```

启动udev

图 7.2.1.1 启动 udev

7.2.1 创建和删除类

自动创建设备节点的工作是在驱动程序的入口函数中完成的，一般在 `cdev_add` 函数后面添加自动创建设备节点相关代码。首先要创建一个 `class` 类，`class` 是个结构体，定义在文件 `include/linux/device.h` 里面。`class_create` 是类创建函数，`class_create` 是个宏定义，内容如下：

示例代码 7.2.1.1 `class_create` 函数

```
546 extern struct class * __must_check __class_create(struct module
547             *owner, const char *name,
548             struct lock_class_key *key);
549 extern void class_destroy(struct class *cls);
550
551 /* This is a #define to keep the compiler from merging different
552 * instances of the __key variable */
553 #define class_create(owner, name) \
554 ({ \
555     static struct lock_class_key __key; \
556     __class_create(owner, name, &__key); \
557 })
```

根据上述代码，将宏 `class_create` 展开以后内容如下：

```
struct class *class_create(struct module *owner, const char *name)
```

`class_create` 一共有两个参数，参数 `owner` 一般为 `THIS_MODULE`，参数 `name` 是类名字。返回值是个指向结构体 `class` 的指针，也就是创建的类。

卸载驱动程序的时候需要删除掉类，类删除函数为 `class_destroy`，函数原型如下：

```
void class_destroy(struct class *cls);
```

参数 `cls` 就是要删除的类。

7.2.2 创建设备

上一小节创建好类以后还不能实现自动创建设备节点，我们还需要在这个类下创建一个设备。使用 `device_create` 函数在类下面创建设备，`device_create` 函数原型如下：

```
struct device *device_create(struct class *class,
                             struct device *parent,
                             dev_t devt,
                             void *drvdata,
                             const char *fmt, ...)
```

`device_create` 是个可变参数函数，参数 `class` 就是设备要到创建哪个类下面；参数 `parent` 是父设备，一般为 `NULL`，也就是没有父设备；参数 `devt` 是设备号；参数 `drvdata` 是设备可能会使用的一些数据，一般为 `NULL`；参数 `fmt` 是设备名字，如果设置 `fmt=xxx` 的话，就会生成 `/dev/xxx` 这个设备文件。返回值就是创建好的设备。

同样的，卸载驱动的时候需要删除掉创建的设备，设备删除函数为 `device_destroy`，函数原型如下：

```
void device_destroy(struct class *cls, dev_t devt)
```

参数 `class` 是要删除的设备所处的类，参数 `devt` 是要删除的设备号。

7.2.3 参考示例

在驱动入口函数里面创建类和设备，在驱动出口函数里面删除类和设备，参考示例如下：

示例代码 7.2.3.1 创建/删除类/设备参考代码

```
1  struct class *class;      /* 类      */
2  struct device *device;    /* 设备    */
3  dev_t devid;             /* 设备号  */
4
5  /* 驱动入口函数 */
6  static int __init xxx_init(void)
7  {
8      /* 创建类 */
9      class = class_create(THIS_MODULE, "xxx");
10     /* 创建设备 */
11     device = device_create(class, NULL, devid, NULL, "xxx");
12     return 0;
13 }
14
15 /* 驱动出口函数 */
16 static void __exit led_exit(void)
17 {
18     /* 删除设备 */
19     device_destroy(newchrled.class, newchrled.devid);
20     /* 删除类 */
21     class_destroy(newchrled.class);
22 }
23
24 module_init(led_init);
25 module_exit(led_exit);
```

7.3 设置文件私有数据

每个硬件设备都有一些属性，比如主设备号(`dev_t`)、类(`class`)、设备(`device`)、开关状态(`state`)等等，在编写驱动的时候你可以将这些属性全部写成变量的形式，如下所示：

示例代码 7.3.1 变量形式的设备属性

```
dev_t devid;          /* 设备号 */
struct cdev cdev;     /* cdev   */
struct class *class;  /* 类     */
struct device *device; /* 设备   */
int major;           /* 主设备号 */
```

```
int minor;                /* 次设备号 */
```

这样写肯定没有问题，但是这样写不专业！对于一个设备的所有属性信息我们最好将其做成一个结构体。编写驱动 `open` 函数的时候将设备结构体作为私有数据添加到设备文件中，如下所示：

示例代码 7.3.2 设备结构体作为私有数据

```
/* 设备结构体 */
1 struct test_dev{
2     dev_t devid;           /* 设备号     */
3     struct cdev cdev;     /* cdev       */
4     struct class *class;  /* 类         */
5     struct device *device; /* 设备       */
6     int major;           /* 主设备号   */
7     int minor;          /* 次设备号   */
8 };
9
10 struct test_dev testdev;
11
12 /* open 函数 */
13 static int test_open(struct inode *inode, struct file *filp)
14 {
15     filp->private_data = &testdev; /* 设置私有数据 */
16     return 0;
17 }
```

在 `open` 函数里面设置好私有数据以后，在 `write`、`read`、`close` 等函数中直接读取 `private_data` 即可得到设备结构体。

7.4 硬件原理图分析

本实验的硬件原理参 6.2 小节即可。

7.5 实验程序编写

本实验对应的例程路径为：[开发板光盘](#)→06、Linux 驱动例程源码→03_newchrled。

本章实验在上一章实验的基础上完成，重点是使用了新的字符设备驱动、设置了文件私有数据、添加了自动创建设备节点相关内容。

7.5.1 LED 灯驱动程序编写

新建名为“03_newchrled”文件夹，然后在 03_newchrled 文件夹里面创建 `vscode` 工程，工作区命名为“newchrled”。工程创建好以后新建 `newchrled.c` 文件，在 `newchrled.c` 里面输入如下内容：

示例代码 7.5.1.1 newchrled.c 文件

```
1 #include <linux/types.h>
2 #include <linux/kernel.h>
3 #include <linux/delay.h>
```

```

4  #include <linux/ide.h>
5  #include <linux/init.h>
6  #include <linux/module.h>
7  #include <linux/errno.h>
8  #include <linux/gpio.h>
9  #include <linux/cdev.h>
10 #include <linux/device.h>
11 // #include <asm/mach/map.h>
12 #include <asm/uaccess.h>
13 #include <asm/io.h>
14 /*****
15 Copyright © ALIENTEK Co., Ltd. 1998-2029. All rights reserved.
16 文件名      : newchrled.c
17 作者        : 正点原子
18 版本        : V1.0
19 描述        : LED 驱动文件。
20 其他        : 无
21 论坛        : www.openedv.com
22 日志        : 初版 V1.0 2022/12/02 正点原子团队创建
23 *****/
24 #define NEWCHRLED_CNT      1          /* 设备号个数 */
25 #define NEWCHRLED_NAME    "newchrled" /* 名字 */
26 #define LEDOFF            0          /* 关灯 */
27 #define LEDON             1          /* 开灯 */
28
29 #define PMU_GRP_BASE      (0xFDC20000)
30 #define PMU_GRP_GPIO0C_IOMUX_L (PMU_GRP_BASE + 0x0010)
31 #define PMU_GRP_GPIO0C_DS_0  (PMU_GRP_BASE + 0X0090)
32
33 #define GPIO0_BASE       (0xFDD60000)
34 #define GPIO0_SWPORT_DR_H (GPIO0_BASE + 0X0004)
35 #define GPIO0_SWPORT_DDR_H (GPIO0_BASE + 0X000C)
36
37 /* 映射后的寄存器虚拟地址指针 */
38 static void __iomem *PMU_GRP_GPIO0C_IOMUX_L_PI;
39 static void __iomem *PMU_GRP_GPIO0C_DS_0_PI;
40 static void __iomem *GPIO0_SWPORT_DR_H_PI;
41 static void __iomem *GPIO0_SWPORT_DDR_H_PI;
42
43 /* newchrled 设备结构体 */
44 struct newchrled_dev{
45     dev_t devid;          /* 设备号 */
46     struct cdev cdev;     /* cdev */

```

```

47     struct class *class;          /* 类          */
48     struct device *device;       /* 设备        */
49     int major;                   /* 主设备号    */
50     int minor;                   /* 次设备号    */
51 };
52
53 struct newchrled_dev newchrled; /* led 设备 */
54
55 /*
56  * @description      : LED 打开/关闭
57  * @param - sta     : LEDON(0) 打开 LED, LEDOFF(1) 关闭 LED
58  * @return          : 无
59  */
60 void led_switch(u8 sta)
61 {
62     u32 val = 0;
63     if(sta == LEDON) {
64         val = readl(GPIO0_SWPORT_DR_H_PI);
65         val &= ~(0X1 << 0); /* bit0 清零*/
66         val |= ((0X1 << 16) | (0X1 << 0)); /* bit16 置 1, 允许写 bit0,
67                                             bit0, 高电平*/
68         writel(val, GPIO0_SWPORT_DR_H_PI);
69     }else if(sta == LEDOFF) {
70         val = readl(GPIO0_SWPORT_DR_H_PI);
71         val &= ~(0X1 << 0); /* bit0 清零*/
72         val |= ((0X1 << 16) | (0X0 << 0)); /* bit16 置 1, 允许写 bit0,
73                                             bit0, 低电平 */
74         writel(val, GPIO0_SWPORT_DR_H_PI);
75     }
76 }
77
78 /*
79  * @description      : 物理地址映射
80  * @return          : 无
81  */
82 void led_remap(void)
83 {
84     PMU_GRF_GPIO0C_IOMUX_L_PI = ioremap(PMU_GRF_GPIO0C_IOMUX_L, 4);
85     PMU_GRF_GPIO0C_DS_0_PI = ioremap(PMU_GRF_GPIO0C_DS_0, 4);
86     GPIO0_SWPORT_DR_H_PI = ioremap(GPIO0_SWPORT_DR_H, 4);
87     GPIO0_SWPORT_DDR_H_PI = ioremap(GPIO0_SWPORT_DDR_H, 4);
88 }
89

```

```

90  /*
91  * @description      : 取消映射
92  * @return           : 无
93  */
94  void led_unmap(void)
95  {
96      /* 取消映射 */
97      iounmap(PMU_GRF_GPIO0C_IOMUX_L_PI);
98      iounmap(PMU_GRF_GPIO0C_DS_0_PI);
99      iounmap(GPIO0_SWPORT_DR_H_PI);
100     iounmap(GPIO0_SWPORT_DDR_H_PI);
101 }
102
103 /*
104 * @description      : 打开设备
105 * @param - inode   : 传递给驱动的 inode
106 * @param - filp    : 设备文件, file 结构体有个叫做 private_data 的成员变量
107 *                  一般在 open 的时候将 private_data 指向设备结构体。
108 * @return           : 0 成功;其他 失败
109 */
110 static int led_open(struct inode *inode, struct file *filp)
111 {
112     filp->private_data = &newchrled; /* 设置私有数据 */
113     return 0;
114 }
115
116 /*
117 * @description      : 从设备读取数据
118 * @param - filp     : 要打开的设备文件(文件描述符)
119 * @param - buf      : 返回给用户空间的数据缓冲区
120 * @param - cnt      : 要读取的数据长度
121 * @param - offt     : 相对于文件首地址的偏移
122 * @return           : 读取的字节数, 如果为负值, 表示读取失败
123 */
124 static ssize_t led_read(struct file *filp, char __user *buf,
125 size_t cnt, loff_t *offt)
126 {
127     return 0;
128 }
129
130 /*
131 * @description      : 向设备写数据
132 * @param - filp     : 设备文件, 表示打开的文件描述符

```

```

133 * @param - buf      : 要写给设备写入的数据
134 * @param - cnt      : 要写入的数据长度
135 * @param - offt     : 相对于文件首地址的偏移
136 * @return           : 写入的字节数, 如果为负值, 表示写入失败
137 */
138 static ssize_t led_write(struct file *filp, const char __user *buf,
139 size_t cnt, loff_t *offt)
140 {
141     int retvalue;
142     unsigned char databuf[1];
143     unsigned char ledstat;
144
145     retvalue = copy_from_user(databuf, buf, cnt);
146     if(retvalue < 0) {
147         printk("kernel write failed!\r\n");
148         return -EFAULT;
149     }
150
151     ledstat = databuf[0];      /* 获取状态值 */
152
153     if(ledstat == LEDON) {
154         led_switch(LEDON);    /* 打开 LED 灯 */
155     } else if(ledstat == LEDOFF) {
156         led_switch(LEDOFF);  /* 关闭 LED 灯 */
157     }
158     return 0;
159 }
160
161 /*
162 * @description      : 关闭/释放设备
163 * @param - filp     : 要关闭的设备文件(文件描述符)
164 * @return           : 0 成功;其他 失败
165 */
166 static int led_release(struct inode *inode, struct file *filp)
167 {
168     return 0;
169 }
170
171 /* 设备操作函数 */
172 static struct file_operations newchrled_fops = {
173     .owner = THIS_MODULE,
174     .open = led_open,
175     .read = led_read,

```

```

176     .write = led_write,
177     .release = led_release,
178 };
179
180 /*
181  * @description      : 驱动出口函数
182  * @param            : 无
183  * @return           : 无
184  */
185 static int __init led_init(void)
186 {
187     u32 val = 0;
188     int ret;
189
190     /* 初始化 LED */
191     /* 1、寄存器地址映射 */
192     led_remap();
193
194     /* 2、设置 GPIO0_C0 为 GPIO 功能。*/
195     val = readl(PMU_GRF_GPIO0C_IOMUX_L_PI);
196     val &= ~(0X7 << 0); /* bit2:0, 清零 */
197     val |= ((0X7 << 16) | (0X0 << 0)); /* bit18:16 置 1, 允许写
bit2:0,
198                                     bit2:0: 0, 用作 GPIO0_C0 */
199     writel(val, PMU_GRF_GPIO0C_IOMUX_L_PI);
200
201     /* 3、设置 GPIO0_C0 驱动能力为 level5 */
202     val = readl(PMU_GRF_GPIO0C_DS_0_PI);
203     val &= ~(0X3F << 0); /* bit5:0 清零*/
204     val |= ((0X3F << 16) | (0X3F << 0)); /* bit21:16 置 1, 允许写
bit5:0,
205                                     bit5:0: 0, 用作 GPIO0_C0 */
206     writel(val, PMU_GRF_GPIO0C_DS_0_PI);
207
208     /* 4、设置 GPIO0_C0 为输出 */
209     val = readl(GPIO0_SWPORT_DDR_H_PI);
210     val &= ~(0X1 << 0); /* bit0 清零*/
211     val |= ((0X1 << 16) | (0X1 << 0)); /* bit16 置 1, 允许写 bit0,
212                                     bit0, 高电平 */
213     writel(val, GPIO0_SWPORT_DDR_H_PI);
214
215     /* 5、设置 GPIO0_C0 为低电平, 关闭 LED 灯。*/
216     val = readl(GPIO0_SWPORT_DR_H_PI);
    
```

```

217     val &= ~(0x1 << 0); /* bit0 清零*/
218     val |= ((0x1 << 16) | (0x0 << 0)); /* bit16 置1, 允许写 bit0,
219                                     bit0, 低电平 */
220     writel(val, GPIO0_SWPORT_DR_H_PI);
221
222     /* 注册字符设备驱动 */
223     /* 1、创建设备号 */
224     if (newchrled.major) { /* 定义了设备号 */
225         newchrled.devid = MKDEV(newchrled.major, 0);
226         ret = register_chrdev_region(newchrled.devid, NEWCHRLED_CNT,
227 NEWCHRLED_NAME);
228         if(ret < 0) {
229             pr_err("cannot register %s char driver [ret=%d]\n",
230 NEWCHRLED_NAME, NEWCHRLED_CNT);
231             goto fail_map;
232         }
233     } else { /* 没有定义设备号 */
234         ret = alloc_chrdev_region(&newchrled.devid, 0,
NEWCHRLED_CNT,
235 NEWCHRLED_NAME); /* 申请设备号 */
236         if(ret < 0) {
237             pr_err("%s Couldn't alloc_chrdev_region, ret=%d\r\n",
238 NEWCHRLED_NAME, ret);
239             goto fail_map;
240         }
241         newchrled.major = MAJOR(newchrled.devid); /* 获取主设备号 */
242         newchrled.minor = MINOR(newchrled.devid); /* 获取次设备号 */
243     }
244     printk("newcheled major=%d,minor=%d\r\n",newchrled.major,
245 newchrled.minor);
246
247     /* 2、初始化 cdev */
248     newchrled.cdev.owner = THIS_MODULE;
249     cdev_init(&newchrled.cdev, &newchrled_fops);
250
251     /* 3、添加一个 cdev */
252     ret = cdev_add(&newchrled.cdev, newchrled.devid,
NEWCHRLED_CNT);
253     if(ret < 0)
254         goto del_unregister;
255
256     /* 4、创建类 */
257     newchrled.class = class_create(THIS_MODULE, NEWCHRLED_NAME);
    
```

```

258     if (IS_ERR(newchrled.class)) {
259         goto del_cdev;
260     }
261
262     /* 5、创建设备 */
263     newchrled.device = device_create(newchrled.class, NULL,
264 newchrled.devid, NULL, NEWCHRLED_NAME);
265     if (IS_ERR(newchrled.device)) {
266         goto destroy_class;
267     }
268
269     return 0;
270
271 destroy_class:
272     class_destroy(newchrled.class);
273 del_cdev:
274     cdev_del(&newchrled.cdev);
275 del_unregister:
276     unregister_chrdev_region(newchrled.devid, NEWCHRLED_CNT);
277 fail_map:
278     led_unmap();
279     return -EIO;
280 }
281
282 /*
283 * @description      : 驱动出口函数
284 * @param            : 无
285 * @return           : 无
286 */
287 static void __exit led_exit(void)
288 {
289     /* 取消映射 */
290     led_unmap();
291
292     /* 注销字符设备驱动 */
293     cdev_del(&newchrled.cdev); /* 删除 cdev */
294     unregister_chrdev_region(newchrled.devid, NEWCHRLED_CNT);
295
296     device_destroy(newchrled.class, newchrled.devid);
297     class_destroy(newchrled.class);
298 }
299
300 module_init(led_init);
    
```

```

301 module_exit(led_exit);
302 MODULE_LICENSE("GPL");
303 MODULE_AUTHOR("ALIENTEK");
304 MODULE_INFO(intree, "Y");
    
```

第 24 行, 宏 NEWCHRLED_CNT 表示设备数量, 在申请设备号或者向 Linux 内核添加字符设备的时候需要设置设备数量, 一般我们一个驱动一个设备, 所以这个宏为 1。

第 25 行, 宏 NEWCHRLED_NAME 表示设备名字, 本实验的设备名为“newchrdev”, 为了方便管理, 所有使用到设备名字的地方统一使用此宏, 当驱动加载成功以后就生成 /dev/newchrled 这个设备文件。

第 44~51 行, 创建设备结构体 newchrled_dev。

第 53 行, 定义一个设备结构体变量 newchrdev, 此变量表示 led 设备。

第 110~114 行, 在 led_open 函数中设置文件的私有数据 private_data 指向 newchrdev。

第 185~280 行, 根据前面讲解的方法在驱动入口函数 led_init 中申请设备号、添加字符设备、创建类和设备。本实验我们采用动态申请设备号的方法, 第 44 行使用 printk 在终端上显示出申请到的主设备号和次设备号。

第 288~298 行, 根据前面讲解的方法, 在驱动出口函数 led_exit 中注销字符新设备、删除类和设备。

总体来说 newchrled.c 文件中的内容不复杂, LED 灯驱动部分的程序和上一章一样。重点就是使用了新的字符设备驱动方法。

7.5.2 编写测试 APP

本章直接使用上一章的测试 APP, 将上一章的 ledApp.c 文件复制到本章实验工程下即可。

7.6 运行测试

7.6.1 编译驱动程序和测试 APP

1、编译驱动程序

编写 Makefile 文件, 本章实验的 Makefile 文件和第五章实验基本一样, 只是将 obj-m 变量的值改为 newchrled.o, Makefile 内容如下所示:

示例代码 7.6.1.1 Makefile 文件

```

1 KERNELDIR := /home/alientek/rk3568_linux_sdk/kernel
.....
4 obj-m := newchrled.o
.....
11 clean:
12 $(MAKE) -C $(KERNELDIR) M=$(CURRENT_PATH) clean
    
```

第 4 行, 设置 obj-m 变量的值为 newchrled.o。

输入如下命令编译出驱动模块文件:

```
make ARCH=arm64 //ARCH=arm64 必须指定, 否则编译会失败
编译成功以后就会生成一个名为“newchrled.ko”的驱动模块文件。
```

2、编译测试 APP

输入如下命令编译测试 ledApp.c 这个测试程序:

```
/opt/atk-dlrk356x-toolchain/bin/aarch64-buildroot-linux-gcc ledApp.c -o ledApp
```

编译成功以后就会生成 ledApp 这个应用程序。

7.6.2 运行测试

先在开发板中输入如下命令关闭 LED0 的心跳灯功能:

```
echo none > /sys/class/leds/work/trigger
```

在 Ubuntu 中将上一小节编译出来的 newchrled.ko 和 ledApp 这两个文件通过 adb 命令发送到开发板的/lib/modules/4.19.232 目录下, 命令如下:

```
adb push newchrled.ko ledApp /lib/modules/4.19.232
```

发送成功以后进入到开发板目录 lib/modules/4.19.232 中, 输入如下命令加载 newchrled.ko 驱动模块:

```
depmod //第一次加载驱动的时候需要运行此命令
```

```
modprobe newchrled //加载驱动
```

驱动加载成功以后会输出申请到的主设备号和次设备号, 如图 7.6.2.1 所示:

```
root@ATK-DLRK356X:/lib/modules/4.19.232# depmod
depmod: WARNING: could not open modules.order at /lib/modules/4.19.232: No such file or directory
depmod: WARNING: could not open modules.builtin at /lib/modules/4.19.232: No such file or directory
root@ATK-DLRK356X:/lib/modules/4.19.232# modprobe newchrled
[ 3151.617879] newcheled major=236,minor=0
root@ATK-DLRK356X:/lib/modules/4.19.232#
```

图 7.6.2.1 申请到的主设备号和次设备号

从图 7.6.2.1 可以看出, 申请到的主设备号为 241, 次设备号为 0。驱动加载成功以后会自动在/dev 目录下创建设备节点文件/dev/newchrdev, 输入如下命令查看/dev/newchrdev 这个设备节点文件是否存在:

```
ls /dev/newchrled -l
```

结果如图 7.6.2.2 所示:

```
root@ATK-DLRK356X:/lib/modules/4.19.232# ls /dev/newchrled -l
crw----- 1 root root 236, 0 Aug  6 01:29 /dev/newchrled
root@ATK-DLRK356X:/lib/modules/4.19.232#
```

图 7.6.2.2 /dev/newchrled 设备节点文件

从图 7.6.2.2 中可以看出, /dev/newchrled 这个设备文件存在, 而且主设备号为 241, 次设备号为 0, 说明设备节点文件创建成功。

驱动节点创建成功以后就可以使用 ledApp 软件来测试驱动是否工作正常, 输入如下命令打开 LED 灯:

```
./ledApp /dev/newchrled 1 //打开 LED 灯
```

输入上述命令以后观察 ATK-DLRK3568 开发板上的红色 LED 灯是否点亮, 如果点亮的话说明驱动工作正常。在输入如下命令关闭 LED 灯:

```
./ledApp /dev/newchrled 0 //关闭 LED 灯
```

输入上述命令以后观察开发板上的红色 LED 灯是否熄灭。如果要卸载驱动的话输入如下命令即可:

```
rmmod newchrled
```

第八章 Linux 设备树

在前面章节中我们多次提到“设备树”这个概念，本章我们就来详细的谈一谈设备树。掌握设备树是 Linux 驱动开发人员必备的技能！因为在新版本的 Linux 中，ARM 相关的驱动全部采用了设备树(也有支持老式驱动的，比较少)，最新出 CPU 在系统启动的时候就支持设备树，比如我们的 RK3568 系列、NXP 的 IMX8 系列等。我们所使用的 Linux 版本为 4.19.232，其支持设备树，所以正点原子 ATK-DLRK3568 开发板的所有 Linux 驱动都是基于设备树的。本章我们就来了解一下设备树的起源、重点学习一下设备树语法。

8.1 什么是设备树?

设备树(Device Tree),将这个词分开就是“设备”和“树”,描述设备树的文件叫做 DTS(Device Tree Source),这个 DTS 文件采用树形结构描述板级设备,也就是开发板上的设备信息,比如 CPU 数量、内存基地址、IIC 接口上接了哪些设备、SPI 接口上接了哪些设备等等,如图 8.1.1 所示:

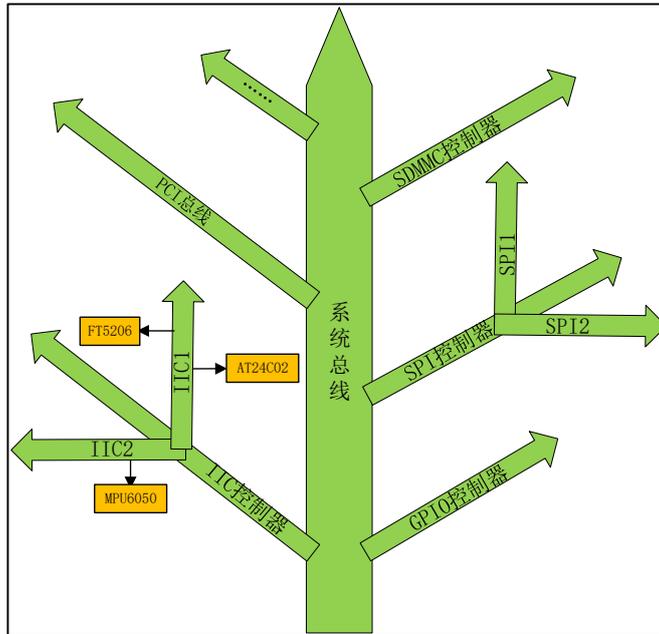


图 8.1.1 设备树结构示意图

在图 8.1.1 中,树的主干就是系统总线, IIC 控制器、GPIO 控制器、SPI 控制器等都是接到系统总线上的分支。IIC 控制器有分为 IIC1 和 IIC2 两种,其中 IIC1 上接了 FT5206 和 AT24C02 这两个 IIC 设备, IIC2 上只接了 MPU6050 这个设备。DTS 文件的主要功能就是按照图 8.1.1 所示的结构来描述板子上的设备信息, DTS 文件描述设备信息是有相应的语法规则要求的,稍后我们会详细的讲解 DTS 语法规则。

在 3.x 版本(具体哪个版本笔者也无从考证)以前的 Linux 内核中 ARM 架构并没有采用设备树。在没有设备树的时候 Linux 是如何描述 ARM 架构中的板级信息呢? 在 Linux 内核源码中大量的 arch/arm/mach-xxx 和 arch/arm/plat-xxx 文件夹, 这些文件夹里面的文件就是对应平台下的板级信息。比如在 arch/arm/mach-s3c24xx/mach-smdk2440.c 中有如下内容(有缩减):

示例代码 8.1.1 mach-smdk2440.c 文件代码段

```

100 static struct s3c2410fb_display smdk2440_lcd_cfg __initdata = {
101
102     .lcdcon5    = S3C2410_LCDCON5_FRM565 |
103                 S3C2410_LCDCON5_INVVLINE |
104                 S3C2410_LCDCON5_INVVFRAME |
105                 S3C2410_LCDCON5_PWREN |
106                 S3C2410_LCDCON5_HWSWP,
107     .....
123 };
124

```

```

125 static struct s3c2410fb_mach_info smdk2440_fb_info __initdata = {
126     .displays    = &smdk2440_lcd_cfg,
127     .num_displays = 1,
128     .default_display = 0,
129     .....
143 };
144
145 static struct platform_device *smdk2440_devices[] __initdata = {
146     &s3c_device_ohci,
147     &s3c_device_lcd,
148     &s3c_device_wdt,
149     &s3c_device_i2c0,
150     &s3c_device_iis,
151 };
    
```

上述代码中第 125 行的结构体变量 `smdk2440_fb_info` 就是描述 SMDK2440 这个开发板上的 LCD 信息的, 结构体指针数组 `smdk2440_devices` 描述的 SMDK2440 这个开发板上的所有平台相关信息。这个仅仅是使用 2440 这个芯片的 SMDK2440 开发板下的 LCD 信息, SMDK2440 开发板还有很多的其他外设硬件和平台硬件信息。使用 2440 这个芯片的板子有很多, 每个板子都有描述相应板级信息的文件, 这仅仅只是一个 2440。随着智能手机的发展, 每年新出的 ARM 架构芯片少说都在数十、甚至数百款, Linux 内核下板级信息文件将会成指数级增长! 这些板级信息文件都是.c 或.h 文件, 都会被硬编码进 Linux 内核中, 导致 Linux 内核“虚胖”。就好比你喜欢吃自助餐, 然后花了 100 多块钱到一家宣传看着很不错的自助餐厅, 结果你想吃的牛排、海鲜、烤肉基本没多少, 全都是一些凉菜、炒面、西瓜、饮料等小吃, 相信你此时肯定会脱口而出一句“F*k!”、“骗子!”。同样的, 当 Linux 之父 linus 看到 ARM 社区向 Linux 内核添加了大量“无用”、冗余的板级信息文件, 不禁的发出了一句“This whole ARM thing is a f*cking pain in the ass”。从此以后 ARM 社区就引入了 PowerPC 等架构已经采用的设备树(Flattened Device Tree), 将这些描述板级硬件信息的内容都从 Linux 内核中分离开来, 用一个专属的文件格式来描述, 这个专属的文件就叫做设备树, 文件扩展名为.dts。一个 SOC 可以作出很多不同的板子, 这些不同的板子肯定是有共同的信息, 将这些共同的信息提取出来作为一个通用的文件, 其他的.dts 文件直接引用这个通用文件即可, 这个通用文件就是.dtsi 文件, 类似于 C 语言中的头文件。一般.dts 描述板级信息(也就是开发板上有哪些 IIC 设备、SPI 设备等), .dtsi 描述 SOC 级信息(也就是 SOC 有几个 CPU、主频是多少、各个外设控制器信息等)。

这个就是设备树的由来, 简而言之就是, Linux 内核中 ARM 架构下有太多的冗余的垃圾板级信息文件, 导致 linus 震怒, 然后 ARM 社区引入了设备树。

8.2 DTS、DTB 和 DTC

上一小节说了, 设备树源文件扩展名为.dts, 但是我们在前面移植 Linux 的时候却一直在使用.dtb 文件, 那么 DTS 和 DTB 这两个文件是什么关系呢? DTS 是设备树源码文件, DTB 是将 DTS 编译以后得到的二进制文件。将.c 文件编译为.o 需要用到 gcc 编译器, 那么将.dts 编译为.dtb 需要用到 DTC 工具! DTC 工具源码在 Linux 内核的 `scripts/dtc` 目录下, `scripts/dtc/Makefile` 文件内容如下:

示例代码 8.2.1 `scripts/dtc/Makefile` 文件代码段

```
4 hostprogs-$(CONFIG_DTC) := dtc
```

```

5 always      := $(hostprogs-y)
6
7 dtc-objs := dtc.o flattree.o fstree.o data.o livetree.o treesource.o
8           srcpos.o checks.o util.o
9 dtc-objs += dtc-lexer.lex.o dtc-parser.tab.o
.....

```

可以看出, DTC 工具依赖于 dtc.c、flattree.c、fstree.c 等文件, 最终编译并链接出 DTC 这个主机文件。如果要编译 DTS 文件的话只需要进入到 Linux 源码根目录下, 然后执行如下命令: (对于 RK3568, 需要指定 ARCH=arm64)

```
make ARCH=arm64 all
```

或者:

```
make ARCH=arm64 dtbs
```

“make ARCH=arm64 all” 命令是编译 Linux 源码中的所有东西, 包括 uImage/zImage, .ko 驱动模块以及设备树, 如果只是编译设备树的话建议使用 “make ARCH=arm64 dtbs” 命令, “make ARCH=arm64 dtbs” 会编译选中的所有设备树文件。如果只要编译指定的某个设备树, 比如我们 ATK-DLRK3568 开发板对应的 “rk3568-atk-evb1-ddr4-v10-linux.dts”, 可以输入如下命令:

```
make ARCH=arm64 rockchip/rk3568-atk-evb1-ddr4-v10-linux.dtb
```

结果如图 8.2.1 所示:

```

allentek@ubuntu:~/rk3568_linux_sdk/kernel$ make ARCH=arm64 rockchip/rk3568-atk-evb1-ddr4-v10-linux.dtb
DTC arch/arm64/boot/dts/rockchip/rk3568-atk-evb1-ddr4-v10-linux.dtb
allentek@ubuntu:~/rk3568_linux_sdk/kernel$

```

图 8.2.1 编译单独的设备树

基于 ARM 架构的 SOC 有很多种, 一种 SOC 又可以制作出很多款板子, 每个板子都有一个对应的 DTS 文件, 那么如何确定编译哪一个 DTS 文件呢? 我们就以 RK3568 这款芯片对应的板子为例来看一下, 打开 arch/arm64/boot/dts/rockchip/Makefile, 有如下内容:

示例代码 8.2.2 arch/arm64/boot/dts/rockchip/Makefile 文件代码段

```

89 dtb-$(CONFIG_ARCH_ROCKCHIP) += rk3566-evb5-lp4x-v10.dtb
90 dtb-$(CONFIG_ARCH_ROCKCHIP) += rk3566-rk817-eink.dtb
91 dtb-$(CONFIG_ARCH_ROCKCHIP) += rk3566-rk817-eink-w6.dtb
92 dtb-$(CONFIG_ARCH_ROCKCHIP) += rk3566-rk817-eink-w103.dtb
93 dtb-$(CONFIG_ARCH_ROCKCHIP) += rk3566-rk817-tablet.dtb
94 dtb-$(CONFIG_ARCH_ROCKCHIP) += rk3566-rk817-tablet-k108.dtb
95 dtb-$(CONFIG_ARCH_ROCKCHIP) += rk3566-rk817-tablet-rkg11.dtb
96 dtb-$(CONFIG_ARCH_ROCKCHIP) += rk3566-rk817-tablet-v10.dtb
97 dtb-$(CONFIG_ARCH_ROCKCHIP) += rk3568-evb1-ddr4-v10.dtb
98 dtb-$(CONFIG_ARCH_ROCKCHIP) += rk3568-evb1-ddr4-v10-android9.dtb
99 dtb-$(CONFIG_ARCH_ROCKCHIP) += rk3568-evb1-ddr4-v10-linux.dtb
100 dtb-$(CONFIG_ARCH_ROCKCHIP) += rk3568-evb1-ddr4-v10-linux-spi-
    nor.dtb
101 dtb-$(CONFIG_ARCH_ROCKCHIP) += rk3568-evb2-lp4x-v10.dtb
102 dtb-$(CONFIG_ARCH_ROCKCHIP) += rk3568-evb2-lp4x-v10-bt1120-to-
    hdm1.dtb
103 dtb-$(CONFIG_ARCH_ROCKCHIP) += rk3568-evb4-lp3-v10.dtb

```

```

104 dtb-$(CONFIG_ARCH_ROCKCHIP) += rk3568-evb5-ddr4-v10.dtb
105 dtb-$(CONFIG_ARCH_ROCKCHIP) += rk3568-evb6-ddr3-v10.dtb
106 dtb-$(CONFIG_ARCH_ROCKCHIP) += rk3568-evb6-ddr3-v10-linux.dtb
107 dtb-$(CONFIG_ARCH_ROCKCHIP) += rk3568-evb6-ddr3-v10-rk628-bt1120-
    to-hdmi.dtb
108 dtb-$(CONFIG_ARCH_ROCKCHIP) += rk3568-evb6-ddr3-v10-rk628-
    rgb2hdmi.dtb
109 dtb-$(CONFIG_ARCH_ROCKCHIP) += rk3568-evb6-ddr3-v10-rk630-bt656-to-
    cvbs.dtb
110 dtb-$(CONFIG_ARCH_ROCKCHIP) += rk3568-evb7-ddr4-v10.dtb
111 dtb-$(CONFIG_ARCH_ROCKCHIP) += rk3568-iotest-ddr3-v10.dtb
112 dtb-$(CONFIG_ARCH_ROCKCHIP) += rk3568-iotest-ddr3-v10-linux.dtb
113 dtb-$(CONFIG_ARCH_ROCKCHIP) += rk3568-nvr-demo-v10.dtb
114 dtb-$(CONFIG_ARCH_ROCKCHIP) += rk3568-nvr-demo-v10-linux.dtb
115 dtb-$(CONFIG_ARCH_ROCKCHIP) += rk3568-nvr-demo-v10-linux-spi-
    nand.dtb
116 dtb-$(CONFIG_ARCH_ROCKCHIP) += rk3568-nvr-demo-v12-linux.dtb
117 dtb-$(CONFIG_ARCH_ROCKCHIP) += rk3568-nvr-demo-v12-linux-spi-
    nand.dtb
118 dtb-$(CONFIG_ARCH_ROCKCHIP) += rk3568-atk-evb1-ddr4-v10-linux.dtb
119 dtb-$(CONFIG_ARCH_ROCKCHIP) += rk630-rk3568-ddr3-v10.dtb
    
```

可以看出, 比如这个 Makefile 下有许多 RK3326、RK3566、RK3568 等不同的.dtb 文件。如果我们使用 RK3568 新做了一个板子, 只需要新建一个此板子对应的.dts 文件, 然后将对应的.dtb 文件名添加到这个 Makefile 下, 这样在编译设备树的时候就会将对应的.dts 编译为二进制的.dtb 文件。

示例代码 8.2.2 中 118 行就是我们在给正点原子的开发板移植 Linux 系统的时候添加的设备树。

8.3 DTS 语法

虽然我们基本上不会从头到尾重写一个.dts 文件, 大多时候是直接在 SOC 厂商提供的.dts 文件上进行修改。但是 DTS 文件语法我们还是需要详细的学习一遍, 因为后续工作中肯定需要修改.dts 文件。大家不要看到要学习新的语法就觉得会很复杂, DTS 语法非常的人性化, 是一种 ASCII 文本文件, 不管是阅读还是修改都很方便。

本节我们就以 rk3568-atk-evb1-ddr4-v10-linux.dts 这个文件为例来讲解一下 DTS 语法。关于设备树详细的语法规则请参考《Devicetree SpecificationV0.2.pdf》和《Power_ePAPR_APPROVED_v1.12.pdf》这两份文档, 此两份文档已经放到了开发板光盘中, 路径为: [开发板光盘](#) → [06、参考资料](#) → [Devicetree SpecificationV0.2.pdf](#) 以及 [Power_ePAPR_APPROVED_v1.12.pdf](#)

8.3.1 .dtsi 头文件

和 C 语言一样, 设备树也支持头文件, 设备树的头文件扩展名为.dtsi。在 rk3568-atk-evb1-ddr4-v10.dtsi 中有如下所示内容:

示例代码 8.3.1.1 rk3568-atk-evb1-ddr4-v10-linux.dts 文件代码段

```
7 #include "rk3568-atk-evb1-ddr4-v10.dtsi"
8 #include "rk3568-linux.dtsi"
```

示例代码 8.3.1.1 中使用“#include”来引用“rk3568-atk-evb1-ddr4-v10.dtsi”和“rk3568-linux.dtsi”这两个.dtsi 头文件。

设备树里面除了可以通过“#include”来引用.dtsi 文件，也可以引用.h 文件头文件，大家打开 rk3568.dtsi 这个文件，找到如下代码：

示例代码 8.3.1.2 rk3568.dtsi 文件代码段

```
1 #include <dt-bindings/clock/rk3568-cru.h>
2 #include <dt-bindings/interrupt-controller/arm-gic.h>
3 #include <dt-bindings/interrupt-controller/irq.h>
4 #include <dt-bindings/pinctrl/rockchip.h>
5 #include <dt-bindings/soc/rockchip,boot-mode.h>
6 #include <dt-bindings/phy/phy.h>
7 #include <dt-bindings/power/rk3568-power.h>
8 #include <dt-bindings/soc/rockchip-system-status.h>
9 #include <dt-bindings/suspend/rockchip-rk3568.h>
10 #include <dt-bindings/thermal/thermal.h>
11 #include "rk3568-dram-default-timing.dtsi"
```

设备树文件不仅可以应用 C 语言里面的.h 头文件，甚至也可以引用.dts 文件。因此在.dts 设备树文件中，可以通过“#include”来引用.h、.dtsi 和.dts 文件。只是，我们在编写设备树头文件的时候最好选择.dtsi 后缀。

一般.dtsi 文件用于描述 SOC 的内部外设信息，比如 CPU 架构、主频、外设寄存器地址范围，比如 UART、IIC 等等。比如 rk3568.dtsi 就是描述 RK3568 芯片本身的外设信息，内容如下：

示例代码 8.3.1.4 rk3568.dtsi 文件代码段

```
6 #include <dt-bindings/clock/rk3568-cru.h>
7 #include <dt-bindings/interrupt-controller/arm-gic.h>
8 #include <dt-bindings/interrupt-controller/irq.h>
.....
18 / {
19     compatible = "rockchip,rk3568";
20
21     interrupt-parent = <&gic>;
22     #address-cells = <2>;
23     #size-cells = <2>;
24
25     aliases {
26         csi2dphy0 = &csi2_dphy0;
27         csi2dphy1 = &csi2_dphy1;
28         csi2dphy2 = &csi2_dphy2;
29         dsi0 = &dsi0;
.....
61     spi3 = &spi3;
```

```

62     spi4 = &sfc; // for U-Boot
63 };
64
65 cpus {
66     #address-cells = <2>;
67     #size-cells = <0>;
68
69     cpu0: cpu@0 {
70         device_type = "cpu";
71         compatible = "arm,cortex-a55";
72         reg = <0x0 0x0>;
73         enable-method = "psci";
74         clocks = <&scmi_clk 0>;
75         operating-points-v2 = <&cpu0_opp_table>;
76         cpu-idle-states = <&CPU_SLEEP>;
77         #cooling-cells = <2>;
78         dynamic-power-coefficient = <187>;
79     };
80
81     cpu1: cpu@100 {
82         device_type = "cpu";
83         compatible = "arm,cortex-a55";
84         reg = <0x0 0x100>;
85         enable-method = "psci";
86         clocks = <&scmi_clk 0>;
87         operating-points-v2 = <&cpu0_opp_table>;
88         cpu-idle-states = <&CPU_SLEEP>;
89     };
90
91     cpu2: cpu@200 {
92         device_type = "cpu";
93         compatible = "arm,cortex-a55";
94         reg = <0x0 0x200>;
95         enable-method = "psci";
96         clocks = <&scmi_clk 0>;
97         operating-points-v2 = <&cpu0_opp_table>;
98         cpu-idle-states = <&CPU_SLEEP>;
99     };
100
101     cpu3: cpu@300 {
102         device_type = "cpu";
103         compatible = "arm,cortex-a55";
104         reg = <0x0 0x300>;
    
```

```

105         enable-method = "psci";
106         clocks = <&scmi_clk 0>;
107         operating-points-v2 = <&cpu0_opp_table>;
108         cpu-idle-states = <&CPU_SLEEP>;
109     };
110
111     idle-states {
112         entry-method = "psci";
113         CPU_SLEEP: cpu-sleep {
114             compatible = "arm,idle-state";
115             local-timer-stop;
116             arm,psci-suspend-param = <0x0010000>;
117             entry-latency-us = <100>;
118             exit-latency-us = <120>;
119             min-residency-us = <1000>;
120         };
121     };
122 };
    
```

示例代码 8.3.1.4 中第 65~122 行就是 RK3568 的 CPU 信息，这个节点信息描述了 RK3568 这颗 SOC 所有的 CPU 信息，一共有 4 个 CPU，也就是 4 核，架构是 cortex-A55。rk3568.dtsi 文件中不仅仅描述了 CPU 信息，RK3568 这颗 SOC 所有的外设都描述的清清楚楚，比如 i2c0~i2c5、uart0~uart9 等等，关于这些设备节点信息的具体内容我们后面具体章节里面再详细的讲解。

8.3.2 设备节点

设备树是采用树形结构来描述板子上的设备信息的文件，每个设备都是一个节点，叫做设备节点，每个节点都通过一些属性信息来描述节点信息，属性就是键-值对。以下文件是结合 RK 官方的设备树缩减出来设备树文件内容：

示例代码 8.3.2.1 设备树模板

```

1 / {
2     compatible = "rockchip,rk3568";
3
4     interrupt-parent = <&gic>;
5     #address-cells = <2>;
6     #size-cells = <2>;
7
8     aliases {
9         serial0 = &uart0;
10    };
11
12    cpus {
13        #address-cells = <2>;
14        #size-cells = <0>;
15
    
```

```

16     cpu0: cpu@0 {
17         device_type = "cpu";
18         compatible = "arm,cortex-a55";
19         reg = <0x0 0x0>;
20         enable-method = "psci";
21         clocks = <&scmi_clk 0>;
22         operating-points-v2 = <&cpu0_opp_table>;
23         cpu-idle-states = <&CPU_SLEEP>;
24         #cooling-cells = <2>;
25         dynamic-power-coefficient = <187>;
26     };
27
28     cpu1: cpu@100 {
29         device_type = "cpu";
30         compatible = "arm,cortex-a55";
31         reg = <0x0 0x100>;
32         enable-method = "psci";
33         clocks = <&scmi_clk 0>;
34         operating-points-v2 = <&cpu0_opp_table>;
35         cpu-idle-states = <&CPU_SLEEP>;
36     };
37
38     cpu2: cpu@200 {
39         device_type = "cpu";
40         compatible = "arm,cortex-a55";
41         reg = <0x0 0x200>;
42         enable-method = "psci";
43         clocks = <&scmi_clk 0>;
44         operating-points-v2 = <&cpu0_opp_table>;
45         cpu-idle-states = <&CPU_SLEEP>;
46     };
47
48     cpu3: cpu@300 {
49         device_type = "cpu";
50         compatible = "arm,cortex-a55";
51         reg = <0x0 0x300>;
52         enable-method = "psci";
53         clocks = <&scmi_clk 0>;
54         operating-points-v2 = <&cpu0_opp_table>;
55         cpu-idle-states = <&CPU_SLEEP>;
56     };
57
58     idle-states {
    
```

```

59         entry-method = "psci";
60         CPU_SLEEP: cpu-sleep {
61             compatible = "arm,idle-state";
62             local-timer-stop;
63             arm,psci-suspend-param = <0x0010000>;
64             entry-latency-us = <100>;
65             exit-latency-us = <120>;
66             min-residency-us = <1000>;
67         };
68     };
69 };
70
71 i2c0: i2c@fdd40000 {
72     compatible = "rockchip,rk3399-i2c";
73     reg = <0x0 0xfdd40000 0x0 0x1000>;
74     clocks = <&pmucru CLK_I2C0>, <&pmucru PCLK_I2C0>;
75     clock-names = "i2c", "pclk";
76     interrupts = <GIC_SPI 46 IRQ_TYPE_LEVEL_HIGH>;
77     pinctrl-names = "default";
78     pinctrl-0 = <&i2c0_xfer>;
79     #address-cells = <1>;
80     #size-cells = <0>;
81     status = "disabled";
82 };
83 };
    
```

第 1 行，“/”是根节点，每个设备树文件只有一个根节点。细心的同学应该会发现，在 rk3568.dtsi 和 rk3568-linux.dtsi 这两个文件都有一个“/”根节点，这样不会出错吗？不会的，因为这两个“/”根节点的内容会合并成一个根节点。

第 8、12 和 71 行，aliases、cpus 和 i2c0 是根节点“/”的三个子节点，在设备树中节点命名格式如下：

node-name@unit-address

其中“node-name”是节点名字，为 ASCII 字符串，节点名字应该能够清晰的描述出节点的功能，比如“uart1”就表示这个节点是 UART1 外设。“unit-address”一般表示设备的地址或寄存器首地址，如果某个节点没有地址或者寄存器的话“unit-address”可以不要，比如“cpus”、“cpu@f00”、“i2c@ff3f0000”。

但是我们在示例代码 8.3.2.1 中我们看到的节点命名却如下所示：

cpu0:cpu@0

上述命令并不是“node-name@unit-address”这样的格式，而是用“:”隔开成了两部分，“:”前面是节点标签(label)，“:”后面的才是节点名字，格式如下所示：

label: node-name@unit-address

引入 label 的目的就是为了方便访问节点，可以直接通过&label 来访问这个节点，比如通过&cpu0 就可以访问“cpu@f00”这个节点，而不需要输入完整的节点名字。再比如节点“i2c0:i2c@ff3f0000”，节点 label 是 i2c0，而节点名字就很长了，为“i2c@ff3f0000”。很明显通过&i2c0

来访问“i2c@ff3f0000”这个节点要方便很多!

第 16 行, cpu0 也是一个节点, 只是 cpu0 是 cpus 的子节点。

每个节点都有不同属性, 不同的属性又有不同的内容, 属性都是键值对, 值可以为空或任意的字节流。设备树源码中常用的几种数据形式如下所示:

1、字符串

```
compatible = "rockchip,rk3568";
```

上述代码设置 compatible 属性的值为字符串“rockchip,rk3568”。

2、32 位无符号整数

```
reg = <0>;
```

上述代码设置 reg 属性的值为 0, reg 的值也可以设置为一组值, 比如:

```
reg = <0 0x123456 100>;
```

3、字符串列表

属性值也可以为字符串列表, 字符串和字符串之间采用“,” 隔开, 如下所示:

```
compatible = "rockchip,rk3568-evb ", "rockchip,rk3568";
```

上述代码设置属性 compatible 的值为“rockchip,RK3568-evb”和“rockchip,rk3568”。

8.3.3 标准属性

节点是由一堆的属性组成, 节点都是具体的设备, 不同的设备需要的属性不同, 用户可以自定义属性。除了用户自定义属性, 有很多属性是标准属性, Linux 下的很多外设驱动都会使用这些标准属性, 本节我们就来学习一下几个常用的标准属性。

1、compatible 属性

compatible 属性也叫做“兼容性”属性, 这是非常重要的一个属性! compatible 属性的值是一个字符串列表, compatible 属性用于将设备和驱动绑定起来。字符串列表用于选择设备所使用的驱动程序, compatible 属性值的格式如下所示:

```
"manufacturer,model"
```

其中 manufacturer 表示厂商, model 一般是模块对应的驱动名字。比如 rk3568-atk-evb1-ddr4-v10.dtsi 中有一个 MIPI 摄像头节点, 这个节点的摄像头芯片采用的 SONY 公司出品的 IMX415, compatible 属性值如下:

```
compatible = "sony,imx415";
```

属性值为“sony,imx415”, 其中‘sony’表示厂商是 sony, 也就是索尼, “imx415”表示驱动模块名字。

compatible 也可以多个属性值。比如:

```
compatible = "ilitek,ili9881d", "simple-panel-dsi";
```

这样我们的设备就有两个属性值, 这个设备首先使用第一个兼容值在 Linux 内核里面查找, 看看能不能找到与之匹配的驱动文件, 如果没有找到的话就使用第二个兼容值查, 以此类推, 直到查找完 compatible 属性中的所有值。

一般驱动程序文件都会有一个 OF 匹配表, 此 OF 匹配表保存着一些 compatible 值, 如果设备节点的 compatible 属性值和 OF 匹配表中的任何一个值相等, 那么就表示设备可以使用这个驱动。比如在文件 imx415.c 中有如下内容:

示例代码 8.3.3.1 imx415.c 文件代码段

```
2598 static const struct of_device_id imx415_of_match[] = {
2599     { .compatible = "sony,imx415" },
```

```
2600     },
2601 };
```

数组 `imx415_of_match` 就是 `imx415.c` 这个驱动文件的匹配表, 此匹配表只有一个匹配值 “`sony,imx415`”。如果在设备树中有哪个节点的 `compatible` 属性值与此相等, 那么这个节点就会使用此驱动文件。

2、model 属性

`model` 属性值也是一个字符串, 一般 `model` 属性描述开发板的名字或者设备模块信息, 比如名字什么的, 比如:

```
model = "Rockchip rk3568 EVB DDR4 V10 Board";
```

3、status 属性

`status` 属性看名字就知道是和设备状态有关的, `status` 属性值也是字符串, 字符串是设备的状态信息, 可选的状态如表 8.3.3.1 所示:

值	描述
“okay”	表明设备是可操作的。
“disabled”	表明设备当前是不可操作的, 但是在未来可以变为可操作的, 比如热插拔设备插入以后。至于 <code>disabled</code> 的具体含义还要看设备的绑定文档。
“fail”	表明设备不可操作, 设备检测到了一系列的错误, 而且设备也不大可能变得可操作。
“fail-sss”	含义和 “fail” 相同, 后面的 <code>sss</code> 部分是检测到的错误内容。

表 8.3.3.1 status 属性值表

4、#address-cells 和 #size-cells 属性

这两个属性的值都是无符号 32 位整形, `#address-cells` 和 `#size-cells` 这两个属性可以在任何拥有子节点的设备中, 用于描述子节点的地址信息。`#address-cells` 属性值决定了子节点 `reg` 属性中地址信息所占用的字长(32 位), `#size-cells` 属性值决定了子节点 `reg` 属性中长度信息所占用的字长(32 位)。`#address-cells` 和 `#size-cells` 表明了子节点应该如何编写 `reg` 属性值, 一般 `reg` 属性都是和地址有关的内容, 和地址相关的信息有两种: 起始地址和地址长度, `reg` 属性的格式为:

```
reg = <address1 length1 address2 length2 address3 length3.....>
```

每个 “`address length`” 组合表示一个地址范围, 其中 `address` 是起始地址, `length` 是地址长度, `#address-cells` 表明 `address` 这个数据所占用的字长, `#size-cells` 表明 `length` 这个数据所占用的字长, 比如:

示例代码 8.3.3.2 #address-cells 和 #size-cells 属性

```
1 cpus {
2     #address-cells = <1>;
3     #size-cells = <0>;
4
5     cpu0: cpu@f00 {
6         device_type = "cpu";
7         compatible = "arm,cortex-a7";
8         reg = <0xf00>;
9         enable-method = "psci";
10        clocks = <&cru ARMCLK>;
11        operating-points-v2 = <&cpu0_opp_table>;
```

```

12     dynamic-power-coefficient = <60>;
13     #cooling-cells = <2>;
14     cpu-idle-states = <&CPU_SLEEP>;
15 };
16 };
17
18 amba {
19     compatible = "simple-bus";
20     #address-cells = <1>;
21     #size-cells = <1>;
22     ranges;
23
24     dmac: dma-controller@ff4e0000 {
25         compatible = "arm,pl330", "arm,primecell";
26         reg = <0xff4e0000 0x4000>;
27         interrupts = <GIC_SPI 1 IRQ_TYPE_LEVEL_HIGH>,
28                 <GIC_SPI 2 IRQ_TYPE_LEVEL_HIGH>;
29         #dma-cells = <1>;
30         clocks = <&cru ACLK_DMACH>;
31         clock-names = "apb_pclk";
32         arm,pl330-periph-burst;
33 };
34 };
    
```

第 2, 3 行, 节点 `cpus` 的 `#address-cells=<1>`, `#size-cells=<0>`, 说明 `cpus` 的子节点 `reg` 属性中起始地址所占用的字长为 1, 地址长度所占用的字长为 0。

第 5 行, 子节点 `cpu0:cpu@0` 的 `reg` 属性值为 `<0xf00>`, 因为父节点设置了 `#address-cells=<1>`, `#size-cells=<0>`, 因此 `address=0xf00`, 没有 `length` 的值, 相当于设置了起始地址, 而没有设置地址长度。

第 20, 21 行, 设置 `amba` 节点 `#address-cells=<1>`, `#size-cells=<1>`, 说明 `amba` 的所有子节点起始地址长度所占用的字长为 1, 地址长度所占用的字长也为 1。

第 26 行, `amba` 子节点 `dmac: dma-controller@ff4e0000` 的 `reg` 属性值为 `<0xff4e0000 0x4000>`, 因为父节点设置了 `#address-cells=<1>`, `#size-cells=<1>`, `address=0xff4e0000`, `length=0x4000`, 相当于设置了起始地址为 `0xff4e0000`, 地址长度为 `0x4000`。

5、reg 属性

`reg` 属性前面已经提到过了, `reg` 属性的值一般是 `(address, length)` 对。 `reg` 属性一般用于描述设备地址空间资源信息或者设备地址信息, 比如某个外设的寄存器地址范围信息, 或者 IIC 器件的设备地址等, 比如在 `rk3568.dtsi` 中有如下内容:

示例代码 8.3.3.3 uart5 节点信息

```

3102     uart5: serial@fe690000 {
3103         compatible = "rockchip,rk3568-uart", "snps,dw-apb-uart";
3104         reg = <0x0 0xfe690000 0x0 0x100>;
3105         interrupts = <GIC_SPI 121 IRQ_TYPE_LEVEL_HIGH>;
3106         clocks = <&cru SCLK_UART5>, <&cru PCLK_UART5>;
    
```

```

3107     clock-names = "baudclk", "apb_pclk";
3108     reg-shift = <2>;
3109     reg-io-width = <4>;
3110     dmas = <&dmac0 10>, <&dmac0 11>;
3111     pinctrl-names = "default";
3112     pinctrl-0 = <&uart5m0_xfer>;
3113     status = "disabled";
3114 };
    
```

uart5 节点描述了 rk3568 系列芯片的 UART5 相关信息，重点是第 3104 行的 reg 属性。由于 uart5 的父节点 “/” 设置了 #address-cells = <1>、#size-cells = <1>，因此 reg 属性中 address=0xff5a0000，length=0x100。查阅《Rockchip RK3568 TRM Part1》可知，RK3568 芯片的 UART5 寄存器首地址为 0xfe690000，长度为 64KB，但是 UART5 的寄存器远远用不了 64KB，0X100 完全够了，这里我们重点是获取 UART5 寄存器首地址。

6、ranges 属性

ranges 属性值可以为空或者按照(child-bus-address,parent-bus-address,length)格式编写的数字矩阵，ranges 是一个地址映射/转换表，ranges 属性每个项目由子地址、父地址和地址空间长度这三部分组成：

child-bus-address: 子总线地址空间的物理地址，由父节点的#address-cells 确定此物理地址所占用的字长。

parent-bus-address: 父总线地址空间的物理地址，同样由父节点的#address-cells 确定此物理地址所占用的字长。

length: 子地址空间的长度，由父节点的#size-cells 确定此地址长度所占用的字长。

如果 ranges 属性值为空值，说明子地址空间和父地址空间完全相同，不需要进行地址转换，对于我们所使用的 RK3568 来说，子地址空间和父地址空间完全相同，因此会在 rk3568.dtsi 中找到大量的值为空的 ranges 属性，如下所示：

示例代码 8.3.3.4 rk3568.dtsi 文件代码段

```

3532     pinctrl: pinctrl {
3533         compatible = "rockchip,rk3568-pinctrl";
3534         rockchip,grf = <&grf>;
3535         rockchip,pmu = <&pmugrf>;
3536         #address-cells = <2>;
3537         #size-cells = <2>;
3538         ranges;
3539         .....
3605 };
    
```

第 2576 行定义了 ranges 属性，但是 ranges 属性值为空。

7、name 属性

name 属性值为字符串，name 属性用于记录节点名字，name 属性已经被弃用，不推荐使用 name 属性，一些老的设备树文件可能会使用此属性。

8、device_type 属性

device_type 属性值为字符串, IEEE 1275 会用到此属性, 用于描述设备的 FCode, 但是设备树没有 FCode, 所以此属性也被抛弃了。此属性只能用于 cpu 节点或者 memory 节点。rk3568.dtsi 的 cpu0 节点用到了此属性, 内容如下所示:

示例代码 8.3.3.6 rk3568.dtsi 文件代码段

```

69  cpu0: cpu@0 {
70      device_type = "cpu";
71      compatible = "arm,cortex-a55";
72      reg = <0x0 0x0>;
73      enable-method = "psci";
74      clocks = <&scmi_clk 0>;
75      operating-points-v2 = <&cpu0_opp_table>;
76      cpu-idle-states = <&CPU_SLEEP>;
77      #cooling-cells = <2>;
78      dynamic-power-coefficient = <187>;
79  };
    
```

关于标准属性就讲解这么多, 其他的比如中断、IIC、SPI 等使用的标准属性等到具体的例程再讲解。

8.3.4 根节点 compatible 属性

每个节点都有 compatible 属性, 根节点 “/” 也不例外, 在 rk3568-atk-evb1-ddr4-v10.dtsi 文件中根节点的 compatible 属性内容如下所示:

示例代码 8.3.4.1 rk3568-atk-evb1-ddr4-v10.dtsi 根节点 compatible 属性

```

15  / {
16      model = "Rockchip RK3568 ATK EVB1 DDR4 V10 Board";
17      compatible = "rockchip,rk3568-evb1-ddr4-v10",
"rockchip,rk3568";
18
19      rk_headset: rk-headset {
    
```

可以看出, compatible 有两个值: “rockchip,rk3568-evb1-ddr4-v10” 和 “rockchip,rk3568”。前面我们说了, 设备节点的 compatible 属性值是为了匹配 Linux 内核中的驱动程序, 那么根节点中的 compatible 属性是为了做什么工作的? 通过根节点的 compatible 属性可以知道我们所使用的设备, 一般第一个值描述了所使用的硬件设备名字, 比如这里使用的是 “rk3568-evb1-ddr4-v10” 这个设备, 第二个值描述了设备所使用的 SOC, 比如这里使用的是 “RK3568” 这颗 SOC。Linux 内核会通过根节点的 compatible 属性查看是否支持此设备, 如果支持的话设备就会启动 Linux 内核。

打开 include/linux/rockchip/cpu.h 文件, 有如下内容:

示例代码 8.3.4.2 判断是否为 RK3568

```

164 static inline bool cpu_is_rk3568(void)
165 {
166     if (rockchip_soc_id)
167         return (rockchip_soc_id & ROCKCHIP_CPU_MASK) ==
ROCKCHIP_CPU_RK3568;
168     return of_machine_is_compatible("rockchip,rk3568");
    
```

```
169 }
86 }
```

函数 `cpu_is_rk3568` 用于判断当前是否为 RK3568, 第 168 行使用 `of_machine_is_compatible` 函数判断根节点 `compatible` 值里面是否有 “rockchip,rk3568”。根据示例代码 8.3.4.1 可知, 根节点的 `compatible` 中有 “rockchip,rk3568”, 所以匹配。

8.3.5 向节点追加或修改内容

产品开发过程中可能面临着频繁的需求更改, 比如第一版硬件上有一个 IIC 接口的六轴芯片 MPU6050, 第二版硬件又要把这个 MPU6050 更换为 MPU9250 等。一旦硬件修改了, 我们就要同步的修改设备树文件, 毕竟设备树是描述板子硬件信息的文件。假设现在有个六轴芯片 `fxls8471`, `fxls8471` 要接到 ATK-DLRK3568 开发板的 I2C1 接口上, 那么相当于需要在 `i2c1` 这个节点上添加一个 `fxls8471` 子节点。先看一下 I2C5 接口对应的节点, 打开文件 `rk3568.dtsi` 文件, 找到如下所示内容:

示例代码 8.3.5.1 i2c5 节点

```
2952 i2c5: i2c@fe5e0000 {
2953     compatible = "rockchip,rk3399-i2c";
2954     reg = <0x0 0xfe5e0000 0x0 0x1000>;
2955     clocks = <&cru CLK_I2C5>, <&cru PCLK_I2C5>;
2956     clock-names = "i2c", "pclk";
2957     interrupts = <GIC_SPI 51 IRQ_TYPE_LEVEL_HIGH>;
2958     pinctrl-names = "default";
2959     pinctrl-0 = <&i2c5m0_xfer>;
2960     #address-cells = <1>;
2961     #size-cells = <0>;
2962     status = "disabled";
2963 };
```

示例代码 8.3.5.1 就是 RK3568 的 `i2c5` 节点, 现在要在 `i2c1` 节点下创建一个子节点, 这个子节点就是 `fxls8471`, 最简单的方法就是在 `i2c5` 下直接添加一个名为 `fxls8471` 的子节点, 如下所示:

示例代码 8.3.5.2 添加 `fxls8471` 子节点

```
2952 i2c5: i2c@fe5e0000 {
2953     compatible = "rockchip,rk3399-i2c";
2954     reg = <0x0 0xfe5e0000 0x0 0x1000>;
2955     clocks = <&cru CLK_I2C5>, <&cru PCLK_I2C5>;
2956     clock-names = "i2c", "pclk";
2957     interrupts = <GIC_SPI 51 IRQ_TYPE_LEVEL_HIGH>;
2958     pinctrl-names = "default";
2959     pinctrl-0 = <&i2c5m0_xfer>;
2960     #address-cells = <1>;
2961     #size-cells = <0>;
2962     status = "disabled";
2963     //fxls8471 子节点
2964     fxls8471@1e {
```

```

2965     compatible = "fsl,fxls8471";
2966     reg = <0x1e>;
2967 };
2968 };
    
```

第 2963~2967 行就是添加的 fxls8471 这个芯片对应的子节点。但是这样会有个问题! i2c5 节点是定义 rk3568.dtsi 文件中的,而 rk3568.dtsi 是共有的设备树头文件,其他所有使用到 rk3568 这颗 SOC 的板子都会引用 rk3568.dtsi 这个文件。直接在 i2c5 节点中添加 fxls8471 就相当于在其他的所有板子上都添加了 fxls8471 这个设备,但是其他的板子并没有这个设备啊!因此,按照示例代码 8.3.5.2 这样写肯定是不行的。

这里就要引入另外一个内容,那就是如何向节点追加数据,我们现在要解决的就是如何向 i2c5 节点追加一个名为 fxls8471 的子节点,而且不能影响到其他使用到 RK3568 的板子。ATK-DLRK3568 开发板使用的设备树文件为 rk3568-atk-evb1-ddr4-v10.dtsi 和 rk3568-linux.dtsi,因此我们需要在 rk3568-atk-evb1-ddr4-v10.dtsi 文件中完成数据追加的内容,方式如下:

示例代码 8.3.5.3 节点追加数据方法

```

1 &i2c5 {
2     /* 要追加或修改的内容 */
3 };
    
```

第 1 行, &i2c5 表示要访问 i2c5 这个 label 所对应的节点,也就是 rk3568.dtsi 中的“i2c5:i2c@fe5e0000”。

第 2 行,花括号内就是要向 i2c5 这个节点添加的内容,包括修改某些属性的值。

正确的做法就是在 rk3568-atk-evb1-ddr4-v10.dtsi 中,向 i2c5 节点追加 fxls8471 相关的信息,如下所示:

示例代码 8.3.5.4 向 i2c1 节点追加数据

```

1 &i2c5 {
2     status = "okay";
3     clock-frequency = <400000>;
4
5     fxls8471@1e {
6         compatible = "fsl,fxls8471";
7         reg = <0x1e>;
8     };
9 }
    
```

示例代码 8.3.5.4 就是向 i2c5 节点添加/修改数据。

第 5~8 行, i2c5 子节点 fxls8471,表示 I2C5 上连接的 fxls8471,“fxls8471”子节点里面描述了 fxls8471 这颗芯片的相关信息。

因为示例代码 8.3.5.4 中的内容是 rk3568-atk-evb1-ddr4-v10.dtsi 这个文件内的,所以不会对使用 RK3568 这颗 SOC 的其他板子造成任何影响。这个就是向节点追加或修改内容,重点就是通过 &label 来访问节点,然后直接在里面编写要追加或者修改的内容。

8.4 创建小型模板设备树

上一节已经对 DTS 的语法做了比较详细的讲解,本节我们就根据前面讲解的语法,从头到尾编写一个小型的设备树文件。当然了,这个小型设备树没有实际的意义,做这个的目的是为了掌握设备树的语法。在实际产品开发中,我们是不需要完完全全的重写一个 .dts 设备树文件,

一般都是使用 SOC 厂商提供好的 .dts 文件，我们只需要在上面根据自己的实际情况做相应的修改即可。在编写设备树之前要先定义一个设备，我们就以 RK3568 这个 SOC 为例，我们需要在设备树里面描述的内容如下：

- ①、这个芯片是由四个 Cortex-A55 架构的 64 位 CPU 组成。
- ②、RK3568 内部 uart2，起始地址为 0xfe660000，大小为 256B(0x100)。
- ③、RK3568 内部 spi0，起始地址为 0xfe610000，大小为 4KB(0x1000)。
- ④、RK3568 内部 i2c5，起始地址为 0xfe5e0000，大小为 4KB(0x1000)。

为了简单起见，我们在设备树里面就实现这些内容即可，首先，搭建一个仅含有根节点“/”的基础的框架，新建一个名为 myfirst.dts 文件，在里面输入如下所示内容：

示例代码 8.4.1 设备树基础框架

```

1 / {
2     compatible = "rockchip,rk3568-evb1-ddr4-v10", "rockchip,rk3568";
3 };
    
```

设备树框架很简单，就一个根节点“/”，根节点里面只有一个 compatible 属性。我们就在这个基础框架上面将上面列出的内容一点点添加进来。

1、添加 cpus 节点

首先添加 CPU 节点，RK3568 采用 Cortex-A55 架构，先添加一个 cpus 节点，在 cpus 节点下添加 cpu0~cpu3 子节点，完成以后如下所示：

示例代码 8.4.2 添加 cpus 节点

```

1 / {
2     compatible = "rockchip,rk3568-evb1-ddr4-v10", "rockchip,rk3568";
3
4     cpus {
5         #address-cells = <2>;
6         #size-cells = <0>;
7
8         /* CPU0 节点 */
9         cpu0: cpu@0 {
10            device_type = "cpu";
11            compatible = "arm,cortex-a55";
12            reg = <0x0 0x0>;
13        };
14
15        /* CPU1 节点 */
16        cpu1: cpu@1 {
17            device_type = "cpu";
18            compatible = "arm,cortex-a55";
19            reg = <0x0 0x100>;
20        };
21
22        /* CPU2 节点 */
23        cpu2: cpu@2 {
24            device_type = "cpu";
    
```

```

25     compatible = "arm,cortex-a55";
26     reg = <0x0 0x200>;
27 };
28
29     /* CPU3 节点 */
30     cpu3: cpu@3 {
31         device_type = "cpu";
32         compatible = "arm,cortex-a55";
33         reg = <0x0 0x300>;
34     };
35 };
36 };
    
```

第 4~35 行, cpus 节点, 此节点用于描述 SOC 内部的所有 CPU, 因为 RK3568 有四个 CPU 核, 所以在 cpus 下添加四个子节点分别为 cpu0、cpu1、cpu2 和 cpu3。注意, 示例代码 8.4.2 里面 cpu0 等子节点内容是参考示例代码, 并没有写全。

2、添加 uart2、spi0 和 i2c5 节点

最后我们在 myfirst.dts 文件中加入 uart2、spi0 和 i2c1 这三个外设控制器的节点。最终的 myfirst.dts 文件内容如下:

示例代码 8.4.3 最终的 myfirst.dts 文件

```

1 / {
2     compatible = "rockchip,rk3568-evb1-ddr4-v10", "rockchip,rk3568";
3
4     cpus {
5         #address-cells = <2>;
6         #size-cells = <0>;
7
8         /* CPU0 节点 */
9         cpu0: cpu@0 {
10            device_type = "cpu";
11            compatible = "arm,cortex-a55";
12            reg = <0x0 0x0>;
13        };
14
15        /* CPU1 节点 */
16        cpu1: cpu@100 {
17            device_type = "cpu";
18            compatible = "arm,cortex-a55";
19            reg = <0x0 0x100>;
20        };
21
22        /* CPU2 节点 */
23        cpu2: cpu@200 {
    
```

```

24     device_type = "cpu";
25     compatible = "arm,cortex-a55";
26     reg = <0x0 0x200>;
27 };
28
29 /* CPU3 节点 */
30 cpu3: cpu@300 {
31     device_type = "cpu";
32     compatible = "arm,cortex-a55";
33     reg = <0x0 0x300>;
34 };
35 };
36
37 /* uart2 节点 */
38 uart2: serial@fe660000 {
39     compatible = "rockchip,rk3568-uart", "snps,dw-apb-uart";
40     reg = <0x0 0xfe660000 0x0 0x100>;
41 };
42
43 /* spi0 节点 */
44 spi0: spi@fe610000 {
45     compatible = "rockchip,rk3568-spi", "rockchip,rk3066-spi";
46     reg = <0x0 0xfe610000 0x0 0x1000>;
47 };
48
49 /* i2c5 节点 */
50 i2c5: i2c@fe5e0000 {
51     compatible = "rockchip,rk3568-i2c", "rockchip,rk3399-i2c";
52     reg = <0x0 0xfe5e0000 0x0 0x1000>;
53 };
54 };
    
```

第 38~41 行, uart2 外设控制器节点。

第 44~47 行, spi0 外设控制器节点。

第 50~53 行, i2c5 外设控制器节点。

到这里, 我们的 myfirst.dts 小型的设备树模板就编辑好了, 基本和 rk3568.dtsi 很像, 可以看做是 rk3568.dtsi 的缩小版。在 myfirst.dts 里面我们仅仅是编写了 RK3568 的外设控制器节点, 节点下的属性并没有写出来。

8.5 设备树在系统中的体现

Linux 内核启动的时候会解析设备树中各个节点的信息, 并且在根文件系统的 /proc/device-tree 目录下根据节点名字创建不同文件夹, 如图 8.5.1 所示:

```

root@ATK-DLRK356X:/# cd /proc/device-tree
root@ATK-DLRK356X:/proc/device-tree# ls
adc-keys                pwm@fe6e0020
#address-cells          pwm@fe6e0030
aliases                 pwm@fe6f0000
arm-pmu                 pwm@fe6f0010
audiopwmout-diff       pwm@fe6f0020
audpwm@fe470000        pwm@fe6f0030
backlight               pwm@fe700000
backlight1              pwm@fe700010
bt-sco                   pwm@fe700020
bt-sound                 pwm@fe700030
    
```

← 根节点 '/' 下所有属性和子节点

图 8.5.1 根节点“/”的属性以及子节点

图 8.5.1 就是目录/proc/device-tree 目录下的内容，/proc/device-tree 目录下是根节点“/”的所有属性和子节点，我们依次来看一下这些属性和子节点。

1、根节点“/”各个属性

在图 8.5.1 中，根节点属性表现为一个个的文件，比如图 8.5.1 中的“#address-cells”、“#size-cells”、“compatible”、“model”和“name”这 5 个文件，它们在设备树中就是根节点的 5 个属性。既然是文件那么肯定可以查看其内容，输入 cat 命令来查看 model 和 compatible 这两个文件的内容，结果如图 8.5.2 所示：

```

root@ATK-DLRK356X:/proc/device-tree# cat model
Rockchip RK3568 ATK EVB1 DDR4 V10 Board
root@ATK-DLRK356X:/proc/device-tree# cat compatible
rockchip,rk3568-evb1-ddr4-v10rockchip,rk3568
    
```

→ model文件内容
→ compatible文件内容

图 8.5.2 model 和 compatible 文件内容

从图 8.5.2 可以看出，文件 model 的内容是“Rockchip RK3568 ATK EVB1 DDR4 V10 Board”，文件 compatible 的内容为“rockchip,rk3568-evb1-ddr4-v10rockchip,rk3568”。打开文件 rk3568-atk-evb1-ddr4-v10.dtsi 查看一下，这不正是根节点“/”的 model 和 compatible 属性值吗！

2、根节点“/”下各子节点

图 8.5.1 中各个文件夹就是根节点“/”的各个子节点，比如“aliases”、“chosen”和“cpus”等等。大家可以查看一下 rk3568-atk-evb1-ddr4-v10.dtsi 和其所引用的所有.dtsi 文件，看看根节点的子节点都有哪些，是否和图 8.5.1 中的一致。

/proc/device-tree 目录就是设备树在根文件系统中的体现，同样是按照树形结构组织的，进入/proc/device-tree/cpus 目录中就可以看到 cpus 节点的所有子节点，如图 8.5.3 所示：

```

root@ATK-DLRK356X:/proc/device-tree/cpus# ls
#address-cells  cpu@100  cpu@300  name
cpu@0           cpu@200  idle-states  #size-cells
    
```

图 8.5.3 cpus 子节点

和根节点“/”一样，图 8.5.3 中的所有文件分别为 soc 节点的属性文件和子节点文件夹。大家可以自行查看一下这些属性文件的内容是否和 rk3568.dtsi 中 cpus 节点的属性值相同。

8.6 特殊节点

在根节点“/”中有两个特殊的子节点：aliases 和 chosen，我们接下来看一下这两个特殊的子节点。

8.6.1 aliases 子节点

打开 rk3568.dtsi 文件，aliases 节点内容如下所示：

示例代码 8.6.1.1 aliases 子节点

```

26 aliases {
27     i2c0 = &i2c0;
28     i2c1 = &i2c1;
29     i2c2 = &i2c2;
30     i2c3 = &i2c3;
    .....
44     dphy0 = &csi_dphy0;
45     dphy1 = &csi_dphy1;
46 };
    
```

单词 `aliases` 的意思是“别名”，因此 `aliases` 节点的主要功能就是定义别名，定义别名的目的就是为了方便访问节点。不过我们一般会在节点命名的时候加上 `label`，然后通过 `&label` 来访问节点，这样也很方便，而且设备树里面大量的使用 `&label` 的形式来访问节点。

8.6.2 chosen 子节点

`chosen` 并不是一个真实的设备，`chosen` 节点主要是为了 `uboot` 向 Linux 内核传递数据，重点是 `bootargs` 参数。一般 `.dts` 文件中 `chosen` 节点通常为空白或者内容很少，`rk3568-linux.dtsi` 中 `chosen` 节点内容如下所示：

示例代码 8.6.2.1 chosen 子节点

```

8 chosen: chosen {
9     bootargs = "earlycon=uart8250,mmio32,0xfe660000
console=ttyFIQ0 root=PARTUUID=614e0000-0000 rw rootwait";
10 };
    
```

从示例代码 8.6.2.1 中可以看出，`chosen` 节点设置了属性“`bootargs`”的值。但是当我们进入到 `/proc/device-tree/chosen` 目录里面查看 `bootargs` 的值，结果如图 8.6.2.1 所示：

```

root@ATK-DLRK356X:/proc/device-tree/chosen# cat bootargs
storagemedia=emmc androidboot.storagemedia=emmc androidboot.mode=normal android
boot.verifiedbootstate=orange androidboot.serialno=fd62623257635519 rw rootwait
earlycon=uart8250,mmio32,0xfe660000 console=ttyFIQ0 root=PARTUUID=614e0000-0000
root@ATK-DLRK356X:/proc/device-tree/chosen#
    
```

图 8.6.2.1 bootargs 文件内容

从图 8.6.2.1 可以看出，`bootargs` 这个文件的内容为“`storagemedia=emmc androidboot.storagemedia=emmc androidboot.mode=normal androidboot.verifiedbootstate=orange androidboot.serialno=fd62623257635519 rw rootwait earlycon=uart8250,mmio32,0xfe660000 console=ttyFIQ0 root=PARTUUID=614e0000-0000`”。和示例代码 8.6.2.1 中的 `bootargs` 不一样，多了前面的“`storagemedia=emmc androidboot.storagemedia=emmc androidboot.mode=normal rootwait`”。

那么问题来了，多出来的“`storagemedia=emmc androidboot.storagemedia=emmc androidboot.mode=normal rootwait`”是怎么来的。

`uboot` 里面的 `bootargs` 环境变量会传递给内核，RK3568 里面的 `bootargs` 值如图 8.6.2.2 所示：

```

boot_targets=mmc1 mmc0 usb0 pxe dhcp
bootargs=storagemedia=emmc androidboot.storagemedia=emmc androidboot.mode=normal
bootcmd=boot_fit;boot_android ${devtype} ${devnum};
    
```

图 8.6.2.2 bootargs 环境变量

从图 8.6.2.2 可以看出，`uboot` 里面的 `bootargs` 环境变量值为：

```
storagemedia=emmc androidboot.storagemedia=emmc androidboot.mode=normal
```

这个不正好就是图 8.6.2.1 中 bootargs 多出来的部分。也就是说最终传递给 Linux 内核的 bootargs 是 uboot 下 bootargs 环境变量的值加设备树里面 bootargs 属性的值。这个通过查看 linux 内核的 cmdline(命令行)参数来得到，如图 8.6.2.3 所示：

```
root@ATK-DLRK356X:/# cat /proc/cmdline
storagemedia=emmc androidboot.storagemedia=emmc androidboot.mode=normal android
boot.verifiedbootstate=orange androidboot.serialno=fd62623257635519 rw rootwait
earlycon=uart8250,mmio32,0xfe660000 console=ttyFIQ0 root=PARTUUID=614e0000-0000
root@ATK-DLRK356X:/#
```

图 8.6.2.3 cmdline 值

从图 8.6.2.3 可以看出，cmdline 的值就和图 8.6.2.1 中的一样，因此是 uboot 来完成 bootargs 环境变量和设备树中的 bootargs 属性值拼接的，然后将其结合体传递给内核。

8.7 绑定信息文档

设备树是用来描述板子上的设备信息的，不同的设备其信息不同，反映到设备树中就是属性不同。那么我们在设备树中添加一个硬件对应的节点的时候从哪里查阅相关的说明呢？在 Linux 内核源码中有详细的 TXT 文档描述了如何添加节点，这些 TXT 文档叫做绑定文档，路径为：Linux 源码目录/Documentation/devicetree/bindings，如图 8.7.1 所示：



图 8.7.1 绑定文档

比如我们现在要想在 RK3568 这颗 SOC 的 I2C 下添加一个节点，那么就可以查看 Documentation/devicetree/bindings/i2c/i2c-rk3x.txt，此文档详细的描述了瑞芯微出品的 SOC 如何在设备树中添加 I2C 设备节点，文档内容如下所示：

```

示例代码 8.7.1 rk3x.txt 文档内容
* Rockchip RK3xxx I2C controller

This driver interfaces with the native I2C controller present in
Rockchip
RK3xxx SoCs.

Required properties :

- reg : Offset and length of the register set for the device
- compatible: should be one of the following:
- "rockchip,rv1108-i2c": for rv1108
    
```

```

- "rockchip,RK3568-i2c": for RK3568
- "rockchip,rk3066-i2c": for rk3066
- "rockchip,rk3188-i2c": for rk3188
- "rockchip,rk3228-i2c": for rk3228
- "rockchip,rk3288-i2c": for rk3288
- "rockchip,rk3328-i2c", "rockchip,rk3399-i2c": for rk3328
- "rockchip,rk3399-i2c": for rk3399
- interrupts : interrupt number
- clocks: See ../clock/clock-bindings.txt
- For older hardware (rk3066, rk3188, rk3228, rk3288):
  - There is one clock that's used both to derive the functional
  clock
    for the device and as the bus clock.
- For newer hardware (rk3399): specified by name
  - "i2c": This is used to derive the functional clock.
  - "pclk": This is the bus clock.
    
```

Required on RK3066, RK3188 :

```

- rockchip,grf : the phandle of the syscon node for the general
register
  file (GRF)
- on those SoCs an alias with the correct I2C bus ID (bit offset in
the GRF)
  is also required.
    
```

Optional properties :

```

- clock-frequency : SCL frequency to use (in Hz). If omitted, 100kHz
is used.
- i2c-scl-rising-time-ns : Number of nanoseconds the SCL signal takes
to rise
  (t(r) in I2C specification). If not specified this is assumed to be
the maximum the specification allows (1000 ns for Standard-mode,
300 ns for Fast-mode) which might cause slightly slower
communication.
- i2c-scl-falling-time-ns : Number of nanoseconds the SCL signal takes
to fall
  (t(f) in the I2C specification). If not specified this is assumed to
be the maximum the specification allows (300 ns) which might cause
slightly slower communication.
- i2c-sda-falling-time-ns : Number of nanoseconds the SDA signal takes
to fall
    
```

(t(f) in the I2C specification). If **not** specified we'll use the SCL value since they are the same in nearly all cases.

Example:

```
aliases {
    i2c0 = &i2c0;
}

i2c0: i2c@2002d000 {
    compatible = "rockchip,rk3188-i2c";
    reg = <0x2002d000 0x1000>;
    interrupts = <GIC_SPI 40 IRQ_TYPE_LEVEL_HIGH>;
    #address-cells = <1>;
    #size-cells = <0>;

    rockchip,grf = <&grf>;

    clock-names = "i2c";
    clocks = <&cru PCLK_I2C0>;

    i2c-scl-rising-time-ns = <800>;
    i2c-scl-falling-time-ns = <100>;
};
```

有时候使用的一些芯片在 `Documentation/devicetree/bindings` 目录下找不到对应的文档，这个时候就要咨询芯片的提供商，让他们给你提供参考的设备树文件。

8.8 设备树常用 OF 操作函数

设备树描述了设备的详细信息，这些信息包括数字类型的、字符串类型的、数组类型的，我们在编写驱动的时候需要获取到这些信息。比如设备树使用 `reg` 属性描述了某个外设的寄存器地址为 `0X02005482`，长度为 `0X400`，我们在编写驱动的时候需要获取到 `reg` 属性的 `0X02005482` 和 `0X400` 这两个值，然后初始化外设。Linux 内核给我们提供了一系列的函数来获取设备树中的节点或者属性信息，这一系列的函数都有一个统一的前缀“`of_`”，所以在很多资料里面也被叫做 OF 函数。这些 OF 函数原型都定义在 `include/linux/of.h` 文件中。

8.8.1 查找节点的 OF 函数

设备都是以节点的形式“挂”到设备树上的，因此要想获取这个设备的其他属性信息，必须先获取到这个设备的节点。Linux 内核使用 `device_node` 结构体来描述一个节点，此结构体定义在文件 `include/linux/of.h` 中，定义如下：

示例代码 8.3.8.1 `device_node` 节点

```
51 struct device_node {
52     const char *name;           /*节点名字      */
53     phandle phandle;
```

```

54     const char *full_name;           /*节点全名          */
55     struct fwnode_handle fwnode;
56
57     struct property *properties;     /*属性              */
58     struct property *deadprops;     /*removed 属性      */
59     struct device_node *parent;     /*父节点            */
60     struct device_node *child;      /*子节点            */
61     struct device_node *sibling;
62 #if defined(CONFIG_OF_KOBJ)
63     struct kobject kobj;
64 #endif
65     unsigned long _flags;
66     void *data;
67 #if defined(CONFIG_SPARC)
68     unsigned int unique_id;
69     struct of_irq_controller *irq_trans;
70 #endif
71 };
    
```

与查找节点有关的 OF 函数有 5 个，我们依次来看一下。

1、of_find_node_by_name 函数

of_find_node_by_name 函数通过节点名字查找指定的节点，函数原型如下：

```

struct device_node *of_find_node_by_name(struct device_node *from,
                                         const char *name);
    
```

函数参数和返回值含义如下：

from: 开始查找的节点，如果为 NULL 表示从根节点开始查找整个设备树。

name: 要查找的节点名字。

返回值: 找到的节点，如果为 NULL 表示查找失败。

2、of_find_node_by_type 函数

of_find_node_by_type 函数通过 device_type 属性查找指定的节点，函数原型如下：

```

struct device_node *of_find_node_by_type(struct device_node *from, const char *type)
    
```

函数参数和返回值含义如下：

from: 开始查找的节点，如果为 NULL 表示从根节点开始查找整个设备树。

type: 要查找的节点对应的 type 字符串，也就是 device_type 属性值。

返回值: 找到的节点，如果为 NULL 表示查找失败。

3、of_find_compatible_node 函数

of_find_compatible_node 函数根据 device_type 和 compatible 这两个属性查找指定的节点，函数原型如下：

```

struct device_node *of_find_compatible_node(struct device_node *from,
                                             const char *type,
                                             const char *compat)
    
```

函数参数和返回值含义如下：

from: 开始查找的节点，如果为 NULL 表示从根节点开始查找整个设备树。

type: 要查找的节点对应的 type 字符串, 也就是 device_type 属性值, 可以为 NULL, 表示忽略掉 device_type 属性。

compat: 要查找的节点所对应的 compatible 属性列表。

返回值: 找到的节点, 如果为 NULL 表示查找失败

4、of_find_matching_node_and_match 函数

of_find_matching_node_and_match 函数通过 of_device_id 匹配表来查找指定的节点, 函数原型如下:

```
struct device_node *of_find_matching_node_and_match(struct device_node *from,
                                                    const struct of_device_id *matches,
                                                    const struct of_device_id **match)
```

函数参数和返回值含义如下:

from: 开始查找的节点, 如果为 NULL 表示从根节点开始查找整个设备树。

matches: of_device_id 匹配表, 也就是在此匹配表里面查找节点。

match: 找到的匹配的 of_device_id。

返回值: 找到的节点, 如果为 NULL 表示查找失败

5、of_find_node_by_path 函数

of_find_node_by_path 函数通过路径来查找指定的节点, 函数原型如下:

```
inline struct device_node *of_find_node_by_path(const char *path)
```

函数参数和返回值含义如下:

path: 带有全路径的节点名, 可以使用节点的别名, 比如 “/backlight” 就是 backlight 这个节点的全路径。

返回值: 找到的节点, 如果为 NULL 表示查找失败

8.8.2 查找父/子节点的 OF 函数

Linux 内核提供了几个查找节点对应的父节点或子节点的 OF 函数, 我们依次来看一下。

1、of_get_parent 函数

of_get_parent 函数用于获取指定节点的父节点(如果有父节点的话), 函数原型如下:

```
struct device_node *of_get_parent(const struct device_node *node)
```

函数参数和返回值含义如下:

node: 要查找的父节点的节点。

返回值: 找到的父节点。

2、of_get_next_child 函数

of_get_next_child 函数用迭代的查找子节点, 函数原型如下:

```
struct device_node *of_get_next_child(const struct device_node *node,
                                      struct device_node *prev)
```

函数参数和返回值含义如下:

node: 父节点。

prev: 前一个子节点, 也就是从哪一个子节点开始迭代的查找下一个子节点。可以设置为 NULL, 表示从第一个子节点开始。

返回值: 找到的下一个子节点。

8.8.3 提取属性值的 OF 函数

节点的属性信息里面保存了驱动所需要的内容, 因此对于属性值的提取非常重要, Linux 内核中使用结构体 `property` 表示属性, 此结构体同样定义在文件 `include/linux/of.h` 中, 内容如下:

示例代码 8.8.3.1 `property` 结构体

```

31 struct property {
32     char    *name;           /* 属性名字          */
33     int     length;         /* 属性长度          */
34     void    *value;         /* 属性值            */
35     struct property *next;   /* 下一个属性        */
36 #if defined(CONFIG_OF_DYNAMIC) || defined(CONFIG_SPARC)
37     unsigned long _flags;
38 #endif
39 #if defined(CONFIG_OF_PROMTREE)
40     unsigned int unique_id;
41 #endif
42 #if defined(CONFIG_OF_KOBJ)
43     struct bin_attribute attr;
44 #endif
45 };
    
```

Linux 内核也提供了提取属性值的 OF 函数, 我们依次来看一下。

1、`of_find_property` 函数

`of_find_property` 函数用于查找指定的属性, 函数原型如下:

```

struct property *of_find_property(const struct device_node *np,
                                const char *name,
                                int *lenp)
    
```

函数参数和返回值含义如下:

np: 设备节点。

name: 属性名字。

lenp: 属性值的字节数

返回值: 找到的属性。

2、`of_property_count_elems_of_size` 函数

`of_property_count_elems_of_size` 函数用于获取属性中元素的数量, 比如 `reg` 属性值是一个数组, 那么使用此函数可以获取到这个数组的大小, 此函数原型如下:

```

int of_property_count_elems_of_size(const struct device_node *np,
                                   const char *propname,
                                   int elem_size)
    
```

函数参数和返回值含义如下:

np: 设备节点。

propname: 需要统计元素数量的属性名字。

elem_size: 元素长度。

返回值: 得到的属性元素数量。

3、of_property_read_u32_index 函数

of_property_read_u32_index 函数用于从属性中获取指定标号的 u32 类型数据值(无符号 32 位), 比如某个属性有多个 u32 类型的值, 那么就可以使用此函数来获取指定标号的数据值, 此函数原型如下:

```
int of_property_read_u32_index(const struct device_node *np,
                             const char *propname,
                             u32 index,
                             u32 *out_value)
```

函数参数和返回值含义如下:

np: 设备节点。

propname: 要读取的属性名字。

index: 要读取的值标号。

out_value: 读取到的值

返回值: 0 读取成功, 负值, 读取失败, -EINVAL 表示属性不存在, -ENODATA 表示没有要读取的数据, -EOVERFLOW 表示属性值列表太小。

4、of_property_read_u8_array 函数

of_property_read_u16_array 函数

of_property_read_u32_array 函数

of_property_read_u64_array 函数

这 4 个函数分别是读取属性中 u8、u16、u32 和 u64 类型的数组数据, 比如大多数的 reg 属性都是数组数据, 可以使用这 4 个函数一次读取出 reg 属性中的所有数据。这四个函数的原型如下:

```
int of_property_read_u8_array(const struct device_node *np,
                             const char *propname,
                             u8 *out_values,
                             size_t sz)
int of_property_read_u16_array(const struct device_node *np,
                              const char *propname,
                              u16 *out_values,
                              size_t sz)
int of_property_read_u32_array(const struct device_node *np,
                               const char *propname,
                               u32 *out_values,
                               size_t sz)
int of_property_read_u64_array(const struct device_node *np,
                               const char *propname,
                               u64 *out_values,
                               size_t sz)
```

函数参数和返回值含义如下:

np: 设备节点。

propname: 要读取的属性名字。

out_value: 读取到的数组值, 分别为 u8、u16、u32 和 u64。

sz: 要读取的数组元素数量。

返回值: 0, 读取成功, 负值, 读取失败, -EINVAL 表示属性不存在, -ENODATA 表示没有要读取的数据, -EOverflow 表示属性值列表太小。

5、of_property_read_u8 函数

of_property_read_u16 函数

of_property_read_u32 函数

of_property_read_u64 函数

有些属性只有一个整形值, 这四个函数就是用于读取这种只有一个整形值的属性, 分别用于读取 u8、u16、u32 和 u64 类型属性值, 函数原型如下:

```
int of_property_read_u8(const struct device_node *np,
                      const char *propname,
                      u8 *out_value)
int of_property_read_u16(const struct device_node *np,
                        const char *propname,
                        u16 *out_value)
int of_property_read_u32(const struct device_node *np,
                        const char *propname,
                        u32 *out_value)
int of_property_read_u64(const struct device_node *np,
                        const char *propname,
                        u64 *out_value)
```

函数参数和返回值含义如下:

np: 设备节点。

propname: 要读取的属性名字。

out_value: 读取到的数组值。

返回值: 0, 读取成功, 负值, 读取失败, -EINVAL 表示属性不存在, -ENODATA 表示没有要读取的数据, -EOverflow 表示属性值列表太小。

6、of_property_read_string 函数

of_property_read_string 函数用于读取属性中字符串值, 函数原型如下:

```
int of_property_read_string(struct device_node *np,
                           const char *propname,
                           const char **out_string)
```

函数参数和返回值含义如下:

np: 设备节点。

propname: 要读取的属性名字。

out_string: 读取到的字符串值。

返回值: 0, 读取成功, 负值, 读取失败。

7、of_n_addr_cells 函数

of_n_addr_cells 函数用于获取#address-cells 属性值, 函数原型如下:

```
int of_n_addr_cells(struct device_node *np)
```

函数参数和返回值含义如下:

np: 设备节点。

返回值: 获取到的#address-cells 属性值。

8、of_n_size_cells 函数

of_n_size_cells 函数用于获取#size-cells 属性值，函数原型如下：

```
int of_n_size_cells(struct device_node *np)
```

函数参数和返回值含义如下：

np: 设备节点。

返回值: 获取到的#size-cells 属性值。

8.8.4 其他常用的 OF 函数

1、of_device_is_compatible 函数

of_device_is_compatible 函数用于查看节点的 compatible 属性是否有包含 name 指定的字符串，也就是检查设备节点的兼容性，函数原型如下：

```
int of_device_is_compatible(const struct device_node *device,
                           const char *name)
```

函数参数和返回值含义如下：

device: 设备节点。

name: 要查看的字符串。

返回值: 0，节点的 compatible 属性中不包含 name 指定的字符串；正数，节点的 compatible 属性中包含 name 指定的字符串。

2、of_get_address 函数

of_get_address 函数用于获取地址相关属性，主要是“reg”或者“assigned-addresses”属性值，函数属性如下：

```
const __be32 *of_get_address(struct device_node *dev,
                             int index,
                             u64 *size,
                             unsigned int *flags)
```

函数参数和返回值含义如下：

dev: 设备节点。

index: 要读取的地址标号。

size: 地址长度。

flags: 参数，比如 IORESOURCE_IO、IORESOURCE_MEM 等

返回值: 读取到的地址数据首地址，为 NULL 的话表示读取失败。

3、of_translate_address 函数

of_translate_address 函数负责将从设备树读取到的物理地址转换为虚拟地址，函数原型如下：

```
u64 of_translate_address(struct device_node *dev,
                        const __be32 *addr)
```

函数参数和返回值含义如下：

dev: 设备节点。

in_addr: 要转换的地址。

返回值: 得到的物理地址，如果为 OF_BAD_ADDR 的话表示转换失败。

4、of_address_to_resource 函数

IIC、SPI、GPIO 等这些外设都有对应的寄存器，这些寄存器其实就是一组内存空间，Linux 内核使用 resource 结构体来描述一段内存空间，“resource”翻译出来就是“资源”，因此用 resource 结构体描述的都是设备资源信息，resource 结构体定义在文件 include/linux/ioport.h 中，定义如下：

示例代码 8.8.4.1 resource 结构体

```

19 struct resource {
20     resource_size_t start;
21     resource_size_t end;
22     const char *name;
23     unsigned long flags;
24     unsigned long desc;
25     struct resource *parent, *sibling, *child;
26 };
    
```

对于 32 位的 SOC 来说，resource_size_t 是 u32 类型的。其中 start 表示开始地址，end 表示结束地址，name 是这个资源的名字，flags 是资源标志位，一般表示资源类型，可选的资源标志定义在文件 include/linux/ioport.h 中，如下所示：

示例代码 8.8.4.2 资源标志

```

1 #define IORESOURCE_BITS           0x000000ff
2 #define IORESOURCE_TYPE_BITS     0x00001f00
3 #define IORESOURCE_IO            0x00000100
4 #define IORESOURCE_MEM          0x00000200
5 #define IORESOURCE_REG          0x00000300
6 #define IORESOURCE_IRQ          0x00000400
7 #define IORESOURCE_DMA          0x00000800
8 #define IORESOURCE_BUS          0x00001000
9 #define IORESOURCE_PREFETCH     0x00002000
10 #define IORESOURCE_READONLY     0x00004000
11 #define IORESOURCE_CACHEABLE   0x00008000
12 #define IORESOURCE_RANGELength  0x00010000
13 #define IORESOURCE_SHADOWABLE   0x00020000
14 #define IORESOURCE_SIZEALIGN    0x00040000
15 #define IORESOURCE_STARTALIGN   0x00080000
16 #define IORESOURCE_MEM_64       0x00100000
17 #define IORESOURCE_WINDOW       0x00200000
18 #define IORESOURCE_MUXED        0x00400000
19 #define IORESOURCE_EXT_TYPE_BITS 0x01000000
20 #define IORESOURCE_SYSRAM       0x01000000
21 #define IORESOURCE_EXCLUSIVE    0x08000000
22 #define IORESOURCE_DISABLED     0x10000000
23 #define IORESOURCE_UNSET        0x20000000
24 #define IORESOURCE_AUTO         0x40000000
25 #define IORESOURCE_BUSY         0x80000000
26 #define IORESOURCE_SYSTEM_RAM   (IORESOURCE_MEM | IORESOURCE_SYSRAM)
    
```

大家一般最常见的资源标志就是 IORESOURCE_MEM、IORESOURCE_REG 和 IORESOURCE_IRQ 等。接下来我们回到 of_address_to_resource 函数，此函数看名字像是从设备树里面提取资源值，但是本质上就是提取 reg 属性值，然后将其转换为 resource 结构体类型，函数原型如下所示

```
int of_address_to_resource(struct device_node *dev,
                           int index,
                           struct resource *r)
```

函数参数和返回值含义如下：

dev: 设备节点。

index: 地址资源标号。

r: 得到的 resource 类型的资源值。

返回值: 0，成功；负值，失败。

5、of_iomap 函数

of_iomap 函数用于直接内存映射，以前我们会通过 ioremap 函数来完成物理地址到虚拟地址的映射，采用设备树以后就可以直接通过 of_iomap 函数来获取内存地址所对应的虚拟地址，不需要使用 ioremap 函数了。当然了，你也可以使用 ioremap 函数来完成物理地址到虚拟地址的内存映射，只是在采用设备树以后，大部分的驱动都使用 of_iomap 函数了。of_iomap 函数本质上也是将 reg 属性中地址信息转换为虚拟地址，如果 reg 属性有多段的话，可以通过 index 参数指定要完成内存映射的是哪一段，of_iomap 函数原型如下：

```
void __iomem *of_iomap(struct device_node *np,
                       int index)
```

函数参数和返回值含义如下：

np: 设备节点。

index: reg 属性中要完成内存映射的段，如果 reg 属性只有一段的话 index 就设置为 0。

返回值: 经过内存映射后的虚拟内存首地址，如果为 NULL 的话表示内存映射失败。

关于设备树常用的 OF 函数就先讲解到这里，Linux 内核中关于设备树的 OF 函数不仅仅只有前面讲的这几个，还有很多 OF 函数我们并没有讲解，这些没有讲解的 OF 函数要结合具体的驱动，比如获取中断号的 OF 函数、获取 GPIO 的 OF 函数等等，这些 OF 函数我们在后面的驱动实验中再详细的讲解。

关于设备树就讲解到这里，关于设备树我们重点要了解一下几点内容：

①、DTS、DTB 和 DTC 之间的区别，如何将 .dts 文件编译为 .dtb 文件。

②、设备树语法，这个是重点，因为在实际工作中我们是需要修改设备树的。

③、设备树的几个特殊子节点。

④、关于设备树的 OF 操作函数，也是重点，因为设备树最终是被驱动文件所使用的，而驱动文件必须要读取设备树中的属性信息，比如内存信息、GPIO 信息、中断信息等等。要想在驱动中读取设备树的属性值，那么就必须使用 Linux 内核提供的众多的 OF 函数。

从下一章开始所以的 Linux 驱动实验都将采用设备树，从最基本的点灯，到复杂的音频、网络或块设备等驱动。将会带领大家由简入深，深度剖析设备树，最终掌握基于设备树的驱动开发技能。

第九章 设备树下的 LED 驱动实验

上一章我们详细的讲解了设备树语法以及在驱动开发中常用的 OF 函数，本章我们就开始第一个基于设备树的 Linux 驱动实验。本章在第七章实验的基础上完成，只是将其驱动开发改为设备树形式而已。

9.1 设备树 LED 驱动原理

在《第七章 新字符设备驱动实验》中，我们直接在驱动文件 `newchrled.c` 中定义有关寄存器物理地址，然后使用 `io_remap` 函数进行内存映射，得到对应的虚拟地址，最后操作寄存器对应的虚拟地址完成对 GPIO 的初始化。本章我们在第七章实验基础上完成，本章我们使用设备树来向 Linux 内核传递相关的寄存器物理地址，Linux 驱动文件使用上一章讲解的 OF 函数从设备树中获取所需的属性值，然后使用获取到的属性值来初始化相关的 IO。本章实验还是比较简单的，本章实验重点内容如下：

- ①、在 `rk3568-atk-evb1-ddr4-v10.dtsi` 文件中创建相应的设备节点。
- ②、编写驱动程序(在第七章实验基础上完成)，获取设备树中的相关属性值。
- ③、使用获取到的有关属性值来初始化 LED 所使用的 GPIO。

9.2 硬件原理图分析

本实验的硬件原理参考 6.2 小节即可。

9.3 实验程序编写

9.3.1 修改设备树文件

在根节点 “/” 下创建一个名为 “`rk3568_led`” 的子节点，打开 `rk3568-atk-evb1-ddr4-v10.dtsi` 文件，在根节点 “/” 最后面输入如下所示内容：

示例代码 9.3.1.1 `rk3568_led` 节点

```
1 rk3568_led {
2     compatible = "atkrk3568-led";
3     status = "okay";
4     reg = <0x0 0xFDC20010 0x0 0x08 /* PMU_GRP_GPIO0C_IOMUX_L */
5           0x0 0xFDC20090 0x0 0x08 /* PMU_GRP_GPIO0C_DS_0 */
6           0x0 0xFDD60004 0x0 0x08 /* GPIO0_SWPORT_DR_H */
7           0x0 0xFDD6000C 0x0 0x08 >; /* GPIO0_SWPORT_DDR_H */
8 };
```

第 2 行，属性 `compatible` 设置 `rk3568_led` 节点兼容为 “`atkrk3568-led`”。

第 3 行，属性 `status` 设置状态为 “`okay`”。

第 4~7 行，`reg` 属性，非常重要！`reg` 属性设置了驱动里面所要使用的寄存器物理地址，比如第 4 行的 “`0x0 0xFDC20010 0x0 0x08`” 表示 RK3568 的 `PMU_GRP_GPIO0C_IOMUX_L` 寄存器，其中寄存器地址为 `0xFDC20010`，长度为 8 个字节。

设备树修改完成以后在 SDK 顶层目录输入如下命令重新编译一下内核：

```
./build.sh kernel
```

编译完成以后得到 `boot.img`，如图 9.3.1.1 所示：

```
allentek@ubuntu:~/rk3568_linux_sdk/kernel$ ls
android          build.config.gki-debug.aarch64  crypto          logo.bmp          scripts
arch             build.config.gki-debug.x86_64   Documentation   logo_kernel.bmp  security
block           build.config.gki_kasan          drivers         MAINTAINERS      sound
boot.img        build.config.gki_kasan.aarch64  firmware       Makefile         System.map
boot.its        build.config.gki_kasan.x86_64   fs             make.sh          tools
build.config.aarch64  build.config.gki_kprobes        include        mm              usr
build.config.allmodconfig  build.config.gki_kprobes.aarch64  init          modules.builtin  virt
build.config.allmodconfig.aarch64  build.config.gki_kprobes.x86_64  ipc          modules.order   vmlinux
build.config.allmodconfig.arm  build.config.gki.x86_64         Kbuild       Module.symvers  vmlinux.o
build.config.allmodconfig.x86_64  build.config.x86_64            Kconfig      net             zboot.img
build.config.arm          built-in.a                     kernel        OWNERS
build.config.common      certs                          kernel.img    README
build.config.gki         COPYING                       lib          resource.img
build.config.gki.aarch64  CREDITS                      LICENSES     samples
allentek@ubuntu:~/rk3568_linux_sdk/kernel$
```

图 9.3.1.1 内核编译出来的 zboot.img

图 9.3.1.1 中 zboot.img 就是编译出来的内核+设备树打包在一起的文件。

烧写方法很简单，分两步：

- ②、开发板进入 LOADER 模式，前提是开发板已经烧写了 uboot，并且 uboot 正常运行。
- ③、打开 RKDevTool 工具，导入配置文件，然后将图 9.3.1.1 中的 boot.img 放到要烧写的目录，然后只烧写 boot.img，详细请看《【正点原子】ATK-DLRK3568 嵌入式 Linux 系统开发手册》第 5.2.3 小节-烧录。

烧写完成以后启动开发板。Linux 启动成功以后进入到 /proc/device-tree/ 目录中查看是否有“rk3568_led”这个节点，结果如图 9.3.1.3 所示：

```
debug@fd904000          qos@fe198000
dfi@fe230000           qos@fe1a8000
display-subsystem     qos@fe1a8080
dmac@fe530000         qos@fe1a8100
dmac@fe550000         reserved-memory
dmc                   rk3568_led
dmcdbg               rk809-sound
dmc-fsp              rkcif_dvp
dmc-opp-table        rkcif_dvp_sditf
dsi@fe060000         rkcif@fdfe0000
dsi@fe070000         rkcif_mipi_lvds
dummy-codec          rkcif_mipi_lvds_sditf
dwmmc@fe000000       rk-headset
dwmmc@fe2b0000       rkisp@fdff0000
dwmmc@fe2c0000       rkisp-vir0
ebc@fdec0000         rkisp-vir1
edp@fe0c0000         rk_rga@fdeb0000
edp-panel            rkscr@fe560000
eink@fdf00000        rkvdec@fdf80200
ethernet@fe010000    rkvinc@fdf40000
ethernet@fe2a0000    rkvinc-opp-table
external-qmac0-clock rm500u-5q
```

图 9.3.1.2 rk3568_led 节点

如果没有“rk3568_led”节点的话请重点检查下面两点：

- ①、检查设备树修改是否成功，也就是 rk3568_led 节点是否为根节点“/”的子节点。
- ②、检查是否使用新的设备树启动的 Linux 内核。

可以进入到图 9.3.1.3 中的 rk3568_led 目录中，查看一下都有哪些属性文件，结果如图 9.3.1.4 所示：

```
root@ATK-DLRK356X:/proc/device-tree/rk3568_led# ls
compatible name reg status
root@ATK-DLRK356X:/proc/device-tree/rk3568_led#
```

图 9.3.1.4 rk3568_led 节点文件

大家可以用 `cat` 命令查看一下 `compatible`、`status` 等属性值是否和我们设置的一致。

9.3.2 LED 灯驱动程序编写

设备树准备好以后就可以编写驱动程序了，本章实验在第七章实验驱动文件 `newchrled.c` 的基础上修改而来。新建名为“04_dtsled”文件夹，然后在 `04_dtsled` 文件夹里面创建 `vscode` 工程，工作区命名为“dtsled”。工程创建好以后新建 `dtsled.c` 文件，在 `dtsled.c` 里面输入如下内容：

示例代码 9.3.2.1 dtsled.c 文件内容

```

1  #include <linux/types.h>
2  #include <linux/kernel.h>
3  #include <linux/delay.h>
4  #include <linux/ide.h>
5  #include <linux/init.h>
6  #include <linux/module.h>
7  #include <linux/errno.h>
8  #include <linux/gpio.h>
9  #include <linux/cdev.h>
10 #include <linux/device.h>
11 #include <linux/of.h>
12 #include <linux/of_address.h>
13 // #include <asm/mach/map.h>
14 #include <asm/uaccess.h>
15 #include <asm/io.h>
16 /*****
17 Copyright © ALIENTEK Co., Ltd. 1998-2029. All rights reserved.
18 文件名      : dtsled.c
19 作者        : 正点原子
20 版本        : V1.0
21 描述        : LED 驱动文件。
22 其他        : 无
23 论坛        : www.openedv.com
24 日志        : 初版 V1.0 2022/12/06 正点原子团队创建
25 *****/
26 #define DTSLED_CNT      1          /* 设备号个数 */
27 #define DTSLED_NAME     "dtsled"  /* 名字 */
28 #define LEDOFF          0          /* 关灯 */
29 #define LEDON           1          /* 开灯 */
30
31 /* 映射后的寄存器虚拟地址指针 */
32 static void __iomem *PMU_GRP_GPIO0C_IOMUX_L_PI;
33 static void __iomem *PMU_GRP_GPIO0C_DS_0_PI;
34 static void __iomem *GPIO0_SWPORT_DR_H_PI;
35 static void __iomem *GPIO0_SWPORT_DDR_H_PI;
36

```

```

37  /* dtsled 设备结构体 */
38  struct dtsled_dev{
39      dev_t devid;          /* 设备号    */
40      struct cdev cdev;    /* cdev    */
41      struct class *class; /* 类      */
42      struct device *device; /* 设备   */
43      int major;          /* 主设备号 */
44      int minor;          /* 次设备号 */
45      struct device_node *nd; /* 设备节点 */
46  };
47
48  struct dtsled_dev dtsled; /* led 设备 */
49
50  /*
51   * @description      : LED 打开/关闭
52   * @param - sta      : LEDON(0) 打开 LED, LEDOFF(1) 关闭 LED
53   * @return           : 无
54   */
55  void led_switch(u8 sta)
56  {
57      u32 val = 0;
58      if(sta == LEDON) {
59          val = readl(GPIO0_SWPORT_DR_H_PI);
60          val &= ~(0X1 << 0); /* bit0 清零*/
61          val |= ((0X1 << 16) | (0X1 << 0)); /* bit16 置 1, 允许写 bit0,
62                                             bit0, 高电平*/
63          writel(val, GPIO0_SWPORT_DR_H_PI);
64      }else if(sta == LEDOFF) {
65          val = readl(GPIO0_SWPORT_DR_H_PI);
66          val &= ~(0X1 << 0); /* bit0 清零*/
67          val |= ((0X1 << 16) | (0X0 << 0)); /* bit16 置 1, 允许写 bit0,
68                                             bit0, 低电平 */
69          writel(val, GPIO0_SWPORT_DR_H_PI);
70      }
71  }
72
73  /*
74   * @description      : 取消映射
75   * @return           : 无
76   */
77  void led_unmap(void)
78  {
79      /* 取消映射 */

```

```

80     iounmap(PMU_GRF_GPIOOC_IOMUX_L_PI);
81     iounmap(PMU_GRF_GPIOOC_DS_0_PI);
82     iounmap(GPIO0_SWPORT_DR_H_PI);
83     iounmap(GPIO0_SWPORT_DDR_H_PI);
84 }
85
86 /*
87  * @description      : 打开设备
88  * @param - inode    : 传递给驱动的 inode
89  * @param - filp     : 设备文件, file 结构体有个叫做 private_data 的成员变量
90  *                   一般在 open 的时候将 private_data 指向设备结构体。
91  * @return           : 0 成功;其他 失败
92  */
93 static int led_open(struct inode *inode, struct file *filp)
94 {
95     filp->private_data = &dtsled; /* 设置私有数据 */
96     return 0;
97 }
98
99 /*
100 * @description      : 从设备读取数据
101 * @param - filp     : 要打开的设备文件 (文件描述符)
102 * @param - buf      : 返回给用户空间的数据缓冲区
103 * @param - cnt      : 要读取的数据长度
104 * @param - offt     : 相对于文件首地址的偏移
105 * @return           : 读取的字节数, 如果为负值, 表示读取失败
106 */
107 static ssize_t led_read(struct file *filp, char __user *buf, size_t
cnt, loff_t *offt)
108 {
109     return 0;
110 }
111
112 /*
113 * @description      : 向设备写数据
114 * @param - filp     : 设备文件, 表示打开的文件描述符
115 * @param - buf      : 要写给设备写入的数据
116 * @param - cnt      : 要写入的数据长度
117 * @param - offt     : 相对于文件首地址的偏移
118 * @return           : 写入的字节数, 如果为负值, 表示写入失败
119 */
120 static ssize_t led_write(struct file *filp, const char __user *buf,
size_t cnt, loff_t *offt)

```

```

121 {
122     int retvalue;
123     unsigned char databuf[1];
124     unsigned char ledstat;
125
126     retvalue = copy_from_user(databuf, buf, cnt);
127     if(retvalue < 0) {
128         printk("kernel write failed!\r\n");
129         return -EFAULT;
130     }
131
132     ledstat = databuf[0];        /* 获取状态值 */
133
134     if(ledstat == LEDON) {
135         led_switch(LEDON);      /* 打开 LED 灯 */
136     } else if(ledstat == LEDOFF) {
137         led_switch(LEDOFF);    /* 关闭 LED 灯 */
138     }
139     return 0;
140 }
141
142 /*
143 * @description      : 关闭/释放设备
144 * @param - filp    : 要关闭的设备文件(文件描述符)
145 * @return          : 0 成功;其他 失败
146 */
147 static int led_release(struct inode *inode, struct file *filp)
148 {
149     return 0;
150 }
151
152 /* 设备操作函数 */
153 static struct file_operations dtsled_fops = {
154     .owner = THIS_MODULE,
155     .open = led_open,
156     .read = led_read,
157     .write = led_write,
158     .release = led_release,
159 };
160
161 /*
162 * @description : 驱动出口函数
163 * @param      : 无
    
```

```

164 * @return      : 无
165 */
166 static int __init led_init(void)
167 {
168     u32 val = 0;
169     int ret;
170     u32 regdata[16];
171     const char *str;
172     struct property *proper;
173
174     /* 获取设备树中的属性数据 */
175     /* 1、获取设备节点: rk3568_led */
176     dtsled.nd = of_find_node_by_path("/rk3568_led");
177     if(dtsled.nd == NULL) {
178         printk("rk3568_led node not find!\r\n");
179         goto fail_find_node;
180     } else {
181         printk("rk3568_led node find!\r\n");
182     }
183
184     /* 2、获取 compatible 属性内容 */
185     proper = of_find_property(dtsled.nd, "compatible", NULL);
186     if(proper == NULL) {
187         printk("compatible property find failed\r\n");
188     } else {
189         printk("compatible = %s\r\n", (char*)proper->value);
190     }
191
192     /* 3、获取 status 属性内容 */
193     ret = of_property_read_string(dtsled.nd, "status", &str);
194     if(ret < 0){
195         printk("status read failed!\r\n");
196     } else {
197         printk("status = %s\r\n", str);
198     }
199
200     /* 4、获取 reg 属性内容 */
201     ret = of_property_read_u32_array(dtsled.nd, "reg", regdata, 16);
202     if(ret < 0) {
203         printk("reg property read failed!\r\n");
204     } else {
205         u8 i = 0;
206         printk("reg data:\r\n");

```

```

207     for(i = 0; i < 16; i++)
208         printk("%#X ", regdata[i]);
209     printk("\r\n");
210 }
211
212 /* 初始化 LED */
213 /* 1、寄存器地址映射 */
214 PMU_GRF_GPIO0C_IOMUX_L_PI = of_iomap(dtsled.nd, 0);
215 PMU_GRF_GPIO0C_DS_0_PI = of_iomap(dtsled.nd, 1);
216 GPIO0_SWPORT_DR_H_PI = of_iomap(dtsled.nd, 2);
217 GPIO0_SWPORT_DDR_H_PI = of_iomap(dtsled.nd, 3);
218
219 /* 2、设置 GPIO0_C0 为 GPIO 功能。*/
220 val = readl(PMU_GRF_GPIO0C_IOMUX_L_PI);
221 val &= ~(0X7 << 0); /* bit2:0, 清零 */
222 val |= ((0X7 << 16) | (0X0 << 0)); /* bit18:16 置 1, 允许写
                                     bit2:0,
                                     bit2:0: 0, 用作 GPIO0_C0 */
223
224 writel(val, PMU_GRF_GPIO0C_IOMUX_L_PI);
225
226 /* 3、设置 GPIO0_C0 驱动能力为 level5 */
227 val = readl(PMU_GRF_GPIO0C_DS_0_PI);
228 val &= ~(0X3F << 0); /* bit5:0 清零*/
229 val |= ((0X3F << 16) | (0X3F << 0)); /* bit21:16 置 1, 允许写
                                     bit5:0,
                                     bit5:0: 0, 用作 GPIO0_C0 */
230
231 writel(val, PMU_GRF_GPIO0C_DS_0_PI);
232
233 /* 4、设置 GPIO0_C0 为输出 */
234 val = readl(GPIO0_SWPORT_DDR_H_PI);
235 val &= ~(0X1 << 0); /* bit0 清零*/
236 val |= ((0X1 << 16) | (0X1 << 0)); /* bit16 置 1, 允许写 bit0,
                                     bit0, 高电平 */
237
238 writel(val, GPIO0_SWPORT_DDR_H_PI);
239
240 /* 5、设置 GPIO0_C0 为低电平, 关闭 LED 灯。*/
241 val = readl(GPIO0_SWPORT_DR_H_PI);
242 val &= ~(0X1 << 0); /* bit0 清零*/
243 val |= ((0X1 << 16) | (0X0 << 0)); /* bit16 置 1, 允许写 bit0,
                                     bit0, 低电平 */
244
245 writel(val, GPIO0_SWPORT_DR_H_PI);
246
247 /* 注册字符设备驱动 */
    
```

```

248     /* 1、创建设备号 */
249     if (dtsled.major) {          /* 定义了设备号 */
250         dtsled.devid = MKDEV(dtsled.major, 0);
251         ret = register_chrdev_region(dtsled.devid, DTSLED_CNT,
                                     DTSLED_NAME);
252
253         if(ret < 0) {
254             pr_err("cannot register %s char driver
255                    [ret=%d]\n",DTSLED_NAME, DTSLED_CNT);
256             goto fail_devid;
257         }
258     } else {                    /* 没有定义设备号 */
259         ret = alloc_chrdev_region(&dtsled.devid, 0, DTSLED_CNT,
                                   DTSLED_NAME); /* 申请设备号 */
260
261         if(ret < 0) {
262             pr_err("%s Couldn't alloc_chrdev_region, ret=%d\r\n",
263                    DTSLED_NAME, ret);
264             goto fail_devid;
265         }
266
267         dtsled.major = MAJOR(dtsled.devid); /* 获取分配号的主设备号 */
268         dtsled.minor = MINOR(dtsled.devid); /* 获取分配号的次设备号 */
269
270     }
271
272     printk("dtsled major=%d,minor=%d\r\n",dtsled.major,
273           dtsled.minor);
274
275     /* 2、初始化 cdev */
276     dtsled.cdev.owner = THIS_MODULE;
277     cdev_init(&dtsled.cdev, &dtsled_fops);
278
279     /* 3、添加一个 cdev */
280     ret = cdev_add(&dtsled.cdev, dtsled.devid, DTSLED_CNT);
281     if(ret < 0)
282         goto del_unregister;
283
284     /* 4、创建类 */
285     dtsled.class = class_create(THIS_MODULE, DTSLED_NAME);
286     if (IS_ERR(dtsled.class)) {
287         goto del_cdev;
288     }
289
290     /* 5、创建设备 */

```

```
285     dtsled.device = device_create(dtsled.class, NULL, dtsled.devid,
                                   NULL, DTSLED_NAME);
286     if (IS_ERR(dtsled.device)) {
287         goto destroy_class;
288     }
289
290     return 0;
291
292 destroy_class:
293     class_destroy(dtsled.class);
294 del_cdev:
295     cdev_del(&dtsled.cdev);
296 del_unregister:
297     unregister_chrdev_region(dtsled.devid, DTSLED_CNT);
298 fail_devid:
299     led_unmap();
300 fail_find_node:
301     return -EIO;
302 }
303
304 /*
305  * @description : 驱动出口函数
306  * @param       : 无
307  * @return      : 无
308  */
309 static void __exit led_exit(void)
310 {
311     /* 取消映射 */
312     led_unmap();
313
314     /* 注销字符设备驱动 */
315     cdev_del(&dtsled.cdev); /* 删除 cdev */
316     unregister_chrdev_region(dtsled.devid, DTSLED_CNT); /* 注销设备号
317 */
318     device_destroy(dtsled.class, dtsled.devid);
319     class_destroy(dtsled.class);
320 }
321
322 module_init(led_init);
323 module_exit(led_exit);
324 MODULE_LICENSE("GPL");
325 MODULE_AUTHOR("ALIENTEK");
```

```
326 MODULE_INFO(intree, "Y");
```

dtsled.c 文件中的内容和第七章的 newchrled.c 文件中的内容基本一样, 只是 dtsled.c 中包含了处理设备树的代码, 我们重点来看一下这部分代码。

第 45 行, 在设备结构体 dtsled_dev 中添加了成员变量 nd, nd 是 device_node 结构体类型指针, 表示设备节点。如果我们要读取设备树某个节点的属性值, 首先要先得到这个节点, 一般在设备结构体中添加 device_node 指针变量来存放这个节点。

第 176~182 行, 通过 of_find_node_by_path 函数得到 rk3568_led 节点, 后续其他的 OF 函数要使用 device_node。

第 185~190 行, 通过 of_find_property 函数获取 rk3568_led 节点的 compatible 属性, 返回值为 property 结构体类型指针变量, property 的成员变量 value 表示属性值。

第 193~198 行, 通过 of_property_read_string 函数获取 rk3568_led 节点的状态属性值。

第 201~210 行, 通过 of_property_read_u32_array 函数获取 rk3568_led 节点的 reg 属性所有值, 并且将获取到的值都存放到 regdata 数组中。第 208 行将获取到的 reg 属性值依次输出到终端上。

第 214~217 行, 使用 of_iomap 函数一次性完成读取 reg 属性以及内存映射, of_iomap 函数是设备树推荐使用的 OF 函数。

9.3.3 编写测试 APP

本章直接使用第七章的测试 APP, 将上一章的 ledApp.c 文件复制到本章实验工程下即可。

9.4.1 编译驱动程序和测试 APP

1、编译驱动程序

编写 Makefile 文件, 本章实验的 Makefile 文件和第五章实验基本一样, 只是将 obj-m 变量的值改为 dtsled.o, Makefile 内容如下所示:

示例代码 9.4.1.1 Makefile 文件

```
1 KERNELDIR := /home/alientek/rk3568_linux_sdk/kernel
.....
4 obj-m := dtsled.o
.....
11 clean:
12 $(MAKE) -C $(KERNELDIR) M=$(CURRENT_PATH) clean
```

第 4 行, 设置 obj-m 变量的值为 dtsled.o。

输入如下命令编译出驱动模块文件:

```
make ARCH=arm64 //ARCH=arm64 必须指定, 否则编译会失败
```

编译成功以后就会生成一个名为“dtsled.ko”的驱动模块文件。

2、编译测试 APP

输入如下命令编译测试 ledApp.c 这个测试程序:

```
/opt/atk-dlrk356x-toolchain/bin/aarch64-buildroot-linux-gnu-gcc ledApp.c -o ledApp
```

编译成功以后就会生成 ledApp 这个应用程序。

9.4.2 运行测试

先在开发板中输入如下命令关闭 LED 的心跳灯功能:

```
echo none > /sys/class/leds/work/trigger
```

在 Ubuntu 中将上一小节编译出来的 dtsled.ko 和 ledApp 这两个文件通过 adb 命令发送到开发板的 /lib/modules/4.19.232 目录下, 命令如下:

```
adb push dtsled.ko ledApp /lib/modules/4.19.232
```

发送成功以后进入到开发板目录 lib/modules/4.19.232 中, 输入如下命令加载 dtsled.ko 驱动模块:

```
depmod //第一次加载驱动的时候需要运行此命令
modprobe dtsled //加载驱动
```

驱动加载成功以后会在终端中输出一些信息, 如图 9.4.2.1 所示:

```
root@ATK-DLRK356X:/lib/modules/4.19.232# modprobe dtsled
[ 2243.932040] rk3568_led node find!
[ 2243.932124] compatible = atkrk3568-led
[ 2243.932134] status = okay
[ 2243.932143] reg data:
[ 2243.932151] 0x0
[ 2243.932154] 0xFDC20010
[ 2243.932159] 0x0
[ 2243.932165] 0x8
[ 2243.932172] 0x0
[ 2243.932177] 0xFDC20090
[ 2243.932183] 0x0
[ 2243.932189] 0x8
[ 2243.932194] root@ATK-DLRK356X:/lib/modules/4.19.232# 32194] 0x0
[ 2243.932199] 0xFDD60004
[ 2243.932205] 0x0
[ 2243.932210] 0x8
[ 2243.932215] 0x0
[ 2243.932220] 0xFDD6000C
[ 2243.932225] 0x0
[ 2243.932230] 0x8
[ 2243.932238]
[ 2243.932862] dtsled major=236,minor=0
root@ATK-DLRK356X:/lib/modules/4.19.232#
```

图 9.4.2.1 驱动加载成功以后输出的信息

从图 9.4.2.1 可以看出, rk3568_led 这个节点找到了, 并且 compatible 属性值为 “atkrk3568-led”, status 属性值为 “okay”, reg 属性的值为 “0x0 0xFDC20010 0x0 0x08 0x0 0xFDC20090 0x0 0x08 0x0 0xFDD60004 0x0 0x08 0x0 0xFDD6000C 0x0 0x08”, 这些都和我们设置的设备树一致。

驱动加载成功以后就可以使用 ledApp 软件来测试驱动是否工作正常, 输入如下命令打开 LED 灯:

```
./ledApp /dev/dtsled 1 //打开 LED 灯
```

输入上述命令以后观察开发板上的红色 LED 灯是否点亮, 如果点亮的话说明驱动工作正常。在输入如下命令关闭 LED 灯:

```
./ledApp /dev/dtsled 0 //关闭 LED 灯
```

输入上述命令以后观察开发板上的红色 LED 灯是否熄灭。如果要卸载驱动的话输入如下命令即可:

```
rmmmod dtsled.ko
```

第十章 pinctrl 和 gpio 子系统实验

上一章我们编写了基于设备树的 LED 驱动，但是驱动的本质还是没变，都是配置 LED 灯所使用的 GPIO 寄存器，驱动开发方式和裸机基本没啥区别。Linux 是一个庞大而完善的系统，尤其是驱动框架，像 GPIO 这种最基本的驱动不可能采用“原始”的裸机驱动开发方式，否则就相当于你买了一辆车，结果每天推着车去上班。Linux 内核提供了 pinctrl 和 gpio 子系统用于 GPIO 驱动，本章我们就来学习一下如何借助 pinctrl 和 gpio 子系统来简化 GPIO 驱动开发。

10.1 pinctrl 子系统

10.1.1 pinctrl 子系统简介

Linux 驱动讲究驱动分离与分层, pinctrl 和 gpio 子系统就是驱动分离与分层思想下的产物, 驱动分离与分层其实就是按照面向对象编程的设计思想而设计的设备驱动框架, 关于驱动的分离与分层我们后面会讲。本来 pinctrl 和 gpio 子系统应该放到驱动分离与分层章节后面讲解, 但是不管什么外设驱动, GPIO 驱动基本都是必须的, 而 pinctrl 和 gpio 子系统又是 GPIO 驱动必须使用的, 所以就将 pinctrl 和 gpio 子系统这一章节提前了。

我们先来回顾一下上一章是怎么初始化 LED 灯所使用的 GPIO, 步骤如下:

①、修改设备树, 添加相应的节点, 节点里面重点是设置 reg 属性, reg 属性包括了 GPIO 相关寄存器。

②、获取 reg 属性中 PMU_GRP_GPIO0C_IOMUX_L、PMU_GRP_GPIO0C_DS_0、GPIO0_SWPORT_DR_H 和 GPIO0_SWPORT_DDR_H 这些寄存器的地址, 并且初始化它们, 这些寄存器用于设置 GPIO0_C0 这个 PIN 的复用功能、上下拉、驱动能力等。

③、在②里面将 GPIO0_C0 这个 PIN 设置为通用输出功能(GPIO), 也就是设置复用功能。

④、在②里面将 GPIO0_C0 这个 PIN 设置为输出模式。

总结一下, ②中完成对 GPIO0_C0 这个 PIN 相关的寄存器地址获取, ③和④设置这个 PIN 的复用功能、上下拉等, 比如将 GPIO0_C0 这个 PIN 设置为通用输出功能。如果使用过 STM32 单片机的话应该都记得, STM32 单片机也是要设置某个 PIN 的复用功能、速度、上下拉等, 然后再设置 PIN 所对应的 GPIO。RK3568 和 STM32 单片机是类似的, 其实对于大多数的 32 位 SOC 而言, 引脚的设置基本都是这两方面:

□、设置 PIN 的复用功能。

□、如果 PIN 复用为 GPIO 功能, 设置 GPIO 相关属性。

因此 Linux 内核针对 PIN 推出了 pinctrl 子系统, 对于 GPIO 的电气属性配置推出了 gpio 子系统。本小节我们来学习 pinctrl 子系统, 下一小节再学习 gpio 子系统。

大多数 SOC 的 pin 都是支持复用的, 比如 RK3568 的 GPIO3_D4 既可以作为普通的 GPIO 使用, 也可以作为 PWM1_M0 引脚、GPU_AVS 引脚、UART0_RX 引脚。此外我们还需要配置 pin 的电气特性, 比如上/下拉、驱动能力等等。传统的配置 pin 的方式就是直接操作相应的寄存器, 但是这种配置方式比较繁琐、而且容易出问题(比如 pin 功能冲突)。pinctrl 子系统就是为了解决这个问题而引入的, pinctrl 子系统主要工作内容如下:

①、获取设备树中 pin 信息。

②、根据获取到的 pin 信息来设置 pin 的复用功能

③、根据获取到的 pin 信息来设置 pin 的电气特性, 如驱动能力。

对于我们使用者来讲, 只需要在设备树里面设置好某个 pin 的相关属性即可, 其他的初始化工作均由 pinctrl 子系统来完成, pinctrl 子系统源码目录为 drivers/pinctrl。

10.1.2 rk3568 的 pinctrl 子系统驱动

1、PIN 配置信息详解

要使用 pinctrl 子系统, 我们需要在设备树里面设置 PIN 的配置信息, 毕竟 pinctrl 子系统要根据你提供的信息来配置 PIN 功能, 一般会在设备树里面创建一个节点来描述 PIN 的配置信息。打开 rk3568.dtsi 文件, 找到一个叫做 pinctrl 的节点, 如下所示:

示例代码 10.1.2.1 pinctrl 节点内容 1

```

3532     pinctrl: pinctrl {
3533         compatible = "rockchip,rk3568-pinctrl";
3534         rockchip,grf = <&grf>;
3535         rockchip,pmu = <&pmugrf>;
3536         #address-cells = <2>;
3537         #size-cells = <2>;
3538         ranges;
3539
3540     gpio0: gpio@fdd60000 {
3541         compatible = "rockchip,gpio-bank";
3542         reg = <0x0 0xfdd60000 0x0 0x100>;
3543         interrupts = <GIC_SPI 33 IRQ_TYPE_LEVEL_HIGH>;
3544         clocks = <&pmucru PCLK_GPIO0>, <&pmucru DBCLK_GPIO0>;
3545
3546         gpio-controller;
3547         #gpio-cells = <2>;
3548         gpio-ranges = <&pinctrl 0 0 32>;
3549         interrupt-controller;
3550         #interrupt-cells = <2>;
3551     };
    .....
3604     };
    
```

第 3536~3537 行, #address-cells 属性值为 2 和 #size-cells 属性值为 2, 也就是说 pinctrl 下的所有子节点的 reg 第一位加第二位是起始地址, 第三位加第四位为长度。

第 3540~3604 行, rk3568 有五组 GPIO: GPIO0~GPIO4, 每组 GPIO 对应的寄存器地址不同。第 3540~3551 行是 GPIO0 这个 GPIO 组对应的子节点, 其中第 2619 行的 reg 属性描述了 GPIO0 对应的寄存器地址, 基地址就是 0XFDD60000, 驱动会得到 GPIO0 的基地址 0XFDD60000, 然后加上偏移就得到了 GPIO0 的其他寄存器地址。

示例代码 10.1.2.1 看起来, 根本没有 PIN 相关的具体配置, 别急! 先打开 rk3568-pinctrl.dtsi 文件, 此文件需要编译内核以后才能得到, 我们能找到如下内容:

示例代码 10.1.2.2 pinctrl 节点内容 2

```

13     &pinctrl {
14         acodec {
15             /omit-if-no-ref/
16             acodec_pins: acodec-pins {
17                 rockchip,pins =
18                     /* acodec_adc_sync */
19                     <1 RK_PB1 5 &pcfg_pull_none>,
20                     /* acodec_adcclk */
21                     <1 RK_PA1 5 &pcfg_pull_none>,
22                     /* acodec_adcdata */
23                     <1 RK_PA0 5 &pcfg_pull_none>,
24                     /* acodec_dac_data1 */
    
```

```

25         <1 RK_PA7 5 &pcfg_pull_none>,
26         /* acodec_dac_datar */
27         <1 RK_PB0 5 &pcfg_pull_none>,
28         /* acodec_dacclk */
29         <1 RK_PA3 5 &pcfg_pull_none>,
30         /* acodec_dacsync */
31         <1 RK_PA5 5 &pcfg_pull_none>;
32     };
33 };
34
35 audiopwm {
36     /omit-if-no-ref/
37     audiopwm_lout: audiopwm-lout {
38         rockchip,pins =
39             /* audiopwm_lout */
40             <1 RK_PA0 4 &pcfg_pull_none>;
41     };
42
43     /omit-if-no-ref/
44     audiopwm_loutn: audiopwm-loutn {
45         rockchip,pins =
46             /* audiopwm_loutn */
47             <1 RK_PA1 6 &pcfg_pull_none>;
48     };
49
50     /omit-if-no-ref/
51     audiopwm_loutp: audiopwm-loutp {
52         rockchip,pins =
53             /* audiopwm_loutp */
54             <1 RK_PA0 6 &pcfg_pull_none>;
55     };
56
57     /omit-if-no-ref/
58     audiopwm_rout: audiopwm-rout {
59         rockchip,pins =
60             /* audiopwm_rout */
61             <1 RK_PA1 4 &pcfg_pull_none>;
62     };
63
64     /omit-if-no-ref/
65     audiopwm_routn: audiopwm-routn {
66         rockchip,pins =
67             /* audiopwm_routn */
    
```

```

68         <1 RK_PA7 4 &pcfg_pull_none>;
69     };
70
71     /omit-if-no-ref/
72     audiopwm_routp: audiopwm-routp {
73         rockchip,pins =
74             /* audiopwm_routp */
75             <1 RK_PA6 4 &pcfg_pull_none>;
76     };
77 };
78
79 bt656 {
80     /omit-if-no-ref/
81     bt656m0_pins: bt656m0-pins {
82         rockchip,pins =
83             /* bt656_clkm0 */
84             <3 RK_PA0 2 &pcfg_pull_none>,
85             /* bt656_d0m0 */
86             <2 RK_PD0 2 &pcfg_pull_none>,
87             /* bt656_d1m0 */
88             <2 RK_PD1 2 &pcfg_pull_none>,
89             /* bt656_d2m0 */
90             <2 RK_PD2 2 &pcfg_pull_none>,
91             /* bt656_d3m0 */
92             <2 RK_PD3 2 &pcfg_pull_none>,
93             /* bt656_d4m0 */
94             <2 RK_PD4 2 &pcfg_pull_none>,
95             /* bt656_d5m0 */
96             <2 RK_PD5 2 &pcfg_pull_none>,
97             /* bt656_d6m0 */
98             <2 RK_PD6 2 &pcfg_pull_none>,
99             /* bt656_d7m0 */
100            <2 RK_PD7 2 &pcfg_pull_none>;
101     };
102
103     /omit-if-no-ref/
104     bt656m1_pins: bt656m1-pins {
105         rockchip,pins =
106             /* bt656_clkm1 */
107             <4 RK_PB4 5 &pcfg_pull_none>,
108             /* bt656_d0m1 */
109             <3 RK_PC6 5 &pcfg_pull_none>,
110             /* bt656_d1m1 */

```

```

111         <3 RK_PC7 5 &pcfg_pull_none>,
112         /* bt656_d2m1 */
113         <3 RK_PD0 5 &pcfg_pull_none>,
114         /* bt656_d3m1 */
115         <3 RK_PD1 5 &pcfg_pull_none>,
116         /* bt656_d4m1 */
117         <3 RK_PD2 5 &pcfg_pull_none>,
118         /* bt656_d5m1 */
119         <3 RK_PD3 5 &pcfg_pull_none>,
120         /* bt656_d6m1 */
121         <3 RK_PD4 5 &pcfg_pull_none>,
122         /* bt656_d7m1 */
123         <3 RK_PD5 5 &pcfg_pull_none>;
124     };
125 };
126
127 bt1120 {
128     /omit-if-no-ref/
129     bt1120_pins: bt1120-pins {
130         rockchip,pins =
131             /* bt1120_clk */
132             <3 RK_PA6 2 &pcfg_pull_none>,
133             /* bt1120_d0 */
134             <3 RK_PA1 2 &pcfg_pull_none>,
135             /* bt1120_d1 */
136             <3 RK_PA2 2 &pcfg_pull_none>,
137             /* bt1120_d2 */
138             <3 RK_PA3 2 &pcfg_pull_none>,
139             /* bt1120_d3 */
140             <3 RK_PA4 2 &pcfg_pull_none>,
141             /* bt1120_d4 */
142             <3 RK_PA5 2 &pcfg_pull_none>,
143             /* bt1120_d5 */
144             <3 RK_PA7 2 &pcfg_pull_none>,
145             /* bt1120_d6 */
146             <3 RK_PB0 2 &pcfg_pull_none>,
147             /* bt1120_d7 */
148             <3 RK_PB1 2 &pcfg_pull_none>,
149             /* bt1120_d8 */
150             <3 RK_PB2 2 &pcfg_pull_none>,
151             /* bt1120_d9 */
152             <3 RK_PB3 2 &pcfg_pull_none>,
153             /* bt1120_d10 */

```

```

154         <3 RK_PB4 2 &pcfg_pull_none>,
155         /* bt1120_d11 */
156         <3 RK_PB5 2 &pcfg_pull_none>,
157         /* bt1120_d12 */
158         <3 RK_PB6 2 &pcfg_pull_none>,
159         /* bt1120_d13 */
160         <3 RK_PC1 2 &pcfg_pull_none>,
161         /* bt1120_d14 */
162         <3 RK_PC2 2 &pcfg_pull_none>,
163         /* bt1120_d15 */
164         <3 RK_PC3 2 &pcfg_pull_none>;
165     };
166 };
167
168     cam {
169         /omit-if-no-ref/
170         cam_clkout0: cam-clkout0 {
171             rockchip,pins =
172                 /* cam_clkout0 */
173                 <4 RK_PA7 1 &pcfg_pull_none>;
174         };
175
176         /omit-if-no-ref/
177         cam_clkout1: cam-clkout1 {
178             rockchip,pins =
179                 /* cam_clkout1 */
180                 <4 RK_PB0 1 &pcfg_pull_none>;
181         };
182     };
183
184     can0 {
185         /omit-if-no-ref/
186         can0m0_pins: can0m0-pins {
187             rockchip,pins =
188                 /* can0_rxm0 */
189                 <0 RK_PB4 2 &pcfg_pull_none>,
190                 /* can0_txm0 */
191                 <0 RK_PB3 2 &pcfg_pull_none>;
192         };
193
194         /omit-if-no-ref/
195         can0m1_pins: can0m1-pins {
196             rockchip,pins =
    
```

```

197         /* can0_rxm1 */
198         <2 RK_PA2 4 &pcfg_pull_none>,
199         /* can0_txm1 */
200         <2 RK_PA1 4 &pcfg_pull_none>;
201     };
202 };
203
204 can1 {
205     /omit-if-no-ref/
206     can1m0_pins: can1m0-pins {
207         rockchip,pins =
208             /* can1_rxm0 */
209             <1 RK_PA0 3 &pcfg_pull_none>,
210             /* can1_txm0 */
211             <1 RK_PA1 3 &pcfg_pull_none>;
212     };
213
214     /omit-if-no-ref/
215     can1m1_pins: can1m1-pins {
216         rockchip,pins =
217             /* can1_rxm1 */
218             <4 RK_PC2 3 &pcfg_pull_none>,
219             /* can1_txm1 */
220             <4 RK_PC3 3 &pcfg_pull_none>;
221     };
222 };
223
224 can2 {
225     /omit-if-no-ref/
226     can2m0_pins: can2m0-pins {
227         rockchip,pins =
228             /* can2_rxm0 */
229             <4 RK_PB4 3 &pcfg_pull_none>,
230             /* can2_txm0 */
231             <4 RK_PB5 3 &pcfg_pull_none>;
232     };
233
234     /omit-if-no-ref/
235     can2m1_pins: can2m1-pins {
236         rockchip,pins =
237             /* can2_rxm1 */
238             <2 RK_PB1 4 &pcfg_pull_none>,
239             /* can2_txm1 */

```

```

240         <2 RK_PB2 4 &pcfg_pull_none>;
241     };
242 };
243
244     cif {
245         /omit-if-no-ref/
246         cif_clk: cif-clk {
247             rockchip,pins =
248                 /* cif_clkout */
249                 <4 RK_PC0 1 &pcfg_pull_none>;
250         };
251
252         /omit-if-no-ref/
253         cif_dvp_clk: cif-dvp-clk {
254             rockchip,pins =
255                 /* cif_clkin */
256                 <4 RK_PC1 1 &pcfg_pull_none>,
257                 /* cif_href */
258                 <4 RK_PB6 1 &pcfg_pull_none>,
259                 /* cif_vsync */
260                 <4 RK_PB7 1 &pcfg_pull_none>;
261         };
262
263     .....
304     };
    
```

示例代码 10.1.2.2 就是向 `pinctrl` 节点追加数据，不同的外设使用的 PIN 不同、其配置也不同，因此一个萝卜一个坑，将某个外设所使用的所有 PIN 都组织在一个子节点里面。示例代码 10.1.2.2 中第 184~242 行的 `can` 子节点就是 rk3568 的三个 CAN 接口所使用的引脚配置，`canm0_pins`、`canm1_pins` 和 `canm2_pins` 分别对应 CAN0、CAN1 和 CAN2 这三个接口。同理第 244 行的子节点 `cif_clk` 就是 CIF0 接口对应时钟的引脚。绑定文档 `Documentation/devicetree/bindings/pinctrl/rockchip,pinctrl.txt` 描述了如何在设备树中设置 rk3568 的 PIN 信息。

每个 `pinctrl` 节点必须至少包含一个子节点来存放 `pinctrl` 相关信息，也就是 `pinctrl` 集，这个集合里面存放当前外设用到哪些引脚(PIN)、复用配置、上下拉、驱动能力等。一般这个存放 `pinctrl` 集的子节点名字是“`rockchip,pins`”。

上面讲了，在“`rockchip,pins`”子节点里面存放外设的引脚描述信息，根据 `rockchip,pinctrl.txt` 文档里面的介绍，引脚复用设置的格式如下：

```
rockchip,pins = <PIN_BANK PIN_BANK_IDX MUX &phandle>
```

一共分为四部分：

1、PIN_BANK

`PIN_BANK` 就是 PIN 所属的组，RK3568 一共有 5 组 PIN：GPIO0~GPIO4，分别对应 0~4，所以如果你要设置 `GPIO0_C0` 这个 PIN，那么 `PIN_BANK` 就是 0。

2、PIN_BANK_IDX

PIN_BANK_IDX 是组内的编号, 以 GPIO0 组为例, 一共有 A0~A7、B0~B7、C0~C7、D0~D7, 这 32 个 PIN。瑞芯微已经给这些 PIN 编了号, 打开 include/dt-bindings/pinctrl/rockchip.h 文件, 有如下定义:

示例代码 10.1.2.3 引脚编号

```

28 #define RK_PA0      0
29 #define RK_PA1      1
30 #define RK_PA2      2
31 #define RK_PA3      3
32 #define RK_PA4      4
33 #define RK_PA5      5
.....
54 #define RK_PD2      26
55 #define RK_PD3      27
56 #define RK_PD4      28
57 #define RK_PD5      29
58 #define RK_PD6      30
59 #define RK_PD7      31
    
```

如果要设置 GPIO0_C0, 那么 PIN_BANK_IDX 就要设置为 RK_PC0。

3、MUX

MUX 就是设置 PIN 的复用功能, 一个 PIN 最多有 16 个复用功能, include/dt-bindings/pinctrl/rockchip.h 文件中有如下内容:

示例代码 10.1.2.4 复用编号

```

61 #define RK_FUNC_GPIO 0
62 #define RK_FUNC_0     0
63 #define RK_FUNC_1     1
64 #define RK_FUNC_2     2
65 #define RK_FUNC_3     3
66 #define RK_FUNC_4     4
67 #define RK_FUNC_5     5
68 #define RK_FUNC_6     6
69 #define RK_FUNC_7     7
70 #define RK_FUNC_8     8
71 #define RK_FUNC_9     9
72 #define RK_FUNC_10    10
73 #define RK_FUNC_11    11
74 #define RK_FUNC_12    12
75 #define RK_FUNC_13    13
76 #define RK_FUNC_14    14
77 #define RK_FUNC_15    15
    
```

上面就是 16 个复用功能编号, 以 GPIO0_C0 为例, 查看 RK3568 数据手册, 可以得到 GPIO0_C0 复用情况如图 10.1.2.1 所示:

2:0	RW	0x0	gpio0c0_sel 3'h0: GPIO0_C0 3'h1: PWM1_M0 3'h2: GPU_AVS 3'h3: UART0_RX
-----	----	-----	---

图 10.1.2.1 GPIO0_C0 复用功能

从图 10.1.2.1 可以看出 GPIO3_D4 有 4 个复用功能:

- 0: GPIO0_C0
- 1: PWM1_M0
- 2: GPU_AVS
- 3: UART0_RX

如果要将 GPIO0_C0 设置为 GPIO 功能, 那么 MUX 就设置 0, 或者 RK_FUNC_GPIO。如果要设置为 PWM1_M0, 那么 MUX 就设置为 1。

4、phandle

最后一个就是 phandle, 用来描述一些引脚的通用配置信息, 打开 scripts/dtc/include-prefixes/arm/rockchip-pinconf.dtsi 文件, 此文件就是 phandle 可选的配置项, 如下所示:

示例代码 10.1.2.5 引脚配置项

```

5  &pinctrl {
6
7      /omit-if-no-ref/
8      pcfg_pull_up: pcfg-pull-up {
9          bias-pull-up;
10     };
11
12     /omit-if-no-ref/
13     pcfg_pull_down: pcfg-pull-down {
14         bias-pull-down;
15     };
16
17     /omit-if-no-ref/
18     pcfg_pull_none: pcfg-pull-none {
19         bias-disable;
20     };
21
22     /omit-if-no-ref/
23     pcfg_pull_none_drv_level_0: pcfg-pull-none-drv-level-0 {
24         bias-disable;
25         drive-strength = <0>;
26     };
27     .....
344 };
    
```

上面的 pcfg_pull_up、pcfg_pull_down、pcfg_pull_none 和 pcfg_pull_none_drv_level_0 就是可使用的配置项。比如 GPIO0_C0 这个 PIN 用作普通的 GPIO, 不需要配置驱动能力, 那么就

可以使用 `pcfg_pull_none`, 也就是不设置上下拉。rockchip-pinconf.dtsi 里面还有很多其他的配置项, 大家自行查阅。

最总, 如果要将 GPIO0_C0 设置为 GPIO 功能, 那么配置就是:

```
rockchip,pins =<0 RK_PC0 RK_FUNC_GPIO &pcfg_pull_none>;
```

10.1.3 设备树中添加 pinctrl 节点模板

我们已经对 pinctrl 有了比较深入的了解, 接下来我们学习一下如何在设备树中添加某个外设的 PIN 信息。比如我们需要将 GPIO0_D1 这个 PIN 复用为 UART2_TX 引脚, pinctrl 节点添加过程如下:

1、创建对应的节点

在 pinctrl 节点下添加一个“uart2”子节点, 然后在 uart2 节点里面在创建一个“uart2m0_xfer: uart2m0-xfer”子节点:

示例代码 10.1.3.1 uart4_pins 设备节点

```
1 &pinctrl {
2   uart2 {
3     /omit-if-no-ref/
4     uart2m0_xfer: uart2m0-xfer {
5       /* 具体的 PIN 信息 */
6     };
7   };
8 };
```

第 3 行的“/omit-if-no-ref/”笔者没有找出说明, 但是瑞芯微官方在每个外设的 pinctrl 引脚设置前都加这一行, 所以这里我们也加上去。

第 4 行, uart2m0_xfer 子节点内就是放置具体的 PIN 配置信息。

2、添加“rockchip,pins”属性

添加一个“rockchip,pins”属性, 这个属性是真正用来描述 PIN 配置信息的, 这里我们只添加 UART2 的 TX 引脚, 所以添加完以后如下所示:

示例代码 10.1.3.2 uart2 设备节点下的 rockchip,pins 属性

```
1 &pinctrl {
2   uart2 {
3     /omit-if-no-ref/
4     uart2m0_xfer: uart2m0-xfer {
5       rockchip,pins =
6         /* uart2_tx_m1 */
7         <0 RK_PD1 1 &pcfg_pull_up>;
8     };
9   };
10 };
```

按道理来讲, 当我们将一个 PIN 用作 GPIO 功能的时候也需要创建对应的 pinctrl 节点, 并且将所用的 PIN 复用为 GPIO 功能, 但是! 对于 RK3568 而言, 如果一个 PIN 用作 GPIO 功能的时候不需要创建对应的 pinctrl 节点!

10.2 gpio 子系统

10.2.1 gpio 子系统简介

上一小节讲解了 pinctrl 子系统, pinctrl 子系统重点是设置 PIN(有的 SOC 叫做 PAD)的复用和电气属性, 如果 pinctrl 子系统将一个 PIN 复用为 GPIO 的话, 那么接下来就要用到 gpio 子系统了。gpio 子系统顾名思义, 就是用于初始化 GPIO 并且提供相应的 API 函数, 比如设置 GPIO 为输入输出, 读取 GPIO 的值等。gpio 子系统的主要目的就是方便驱动开发者使用 gpio, 驱动开发者在设备树中添加 gpio 相关信息, 然后就可以在驱动程序中使用 gpio 子系统提供的 API 函数来操作 GPIO, Linux 内核向驱动开发者屏蔽掉了 GPIO 的设置过程, 极大的方便了驱动开发者使用 GPIO。

10.2.2 rk3568 的 gpio 子系统驱动

1、设备树中的 gpio 信息

首先肯定是 GPIO 控制器的节点信息, 以 GPIO0_C0 这个引脚所在的 GPIO3 为例, 打开 rk3568.dtsi, 在里面找到如下所示内容:

示例代码 10.2.2.1 gpio3 控制器节点

```

3540 gpio0: gpio@fdd60000 {
3541     compatible = "rockchip,gpio-bank";
3542     reg = <0x0 0xfdd60000 0x0 0x100>;
3543     interrupts = <GIC_SPI 33 IRQ_TYPE_LEVEL_HIGH>;
3544     clocks = <&pmucru PCLK_GPIO0>, <&pmucru DBCLK_GPIO0>;
3545
3546     gpio-controller;
3547     #gpio-cells = <2>;
3548     gpio-ranges = <&pinctrl 0 0 32>;
3549     interrupt-controller;
3550     #interrupt-cells = <2>;
3551 };
    
```

示例代码 10.2.2.1 就是 GPIO0 的控制器信息, 属于 pinctrl 的子节点, 绑定文档 [Documentation/devicetree/bindings/gpio/gpio.txt](https://www.kernel.org/doc/html/latest/devicetree/bindings/gpio/gpio.txt) 详细描述了 gpio 控制器节点各个属性信息。

第 3541 行, compatible 属性值为 “rockchip,gpio-bank”, 所以在 linux 内核中搜索这个字符串就可以找到对应的 GPIO 驱动源文件, 为 drivers/pinctrl/pinctrl-rockchip.c。

第 3542 行, reg 属性设置了 GPIO0 控制器的寄存器基地址为 0XFDD60000。

第 3543 行, interrupts 属性描述 GPIO0 控制器对应的中断信息。

第 3544 行, clocks 属性指定这个 GPIO0 控制器的时钟。

第 3546 行, “gpio-controller” 表示 gpio0 节点是个 GPIO 控制器, 每个 GPIO 控制器节点必须包含 “gpio-controller” 属性。

第 3547 行, “#gpio-cells” 属性和 “#address-cells” 类似, #gpio-cells 应该为 2, 表示一共有两个 cell, 第一个 cell 为 GPIO 编号, 比如 “&gpio0 RK_PC0” 就表示 GPIO0_C0。第二个 cell 表示 GPIO 极性, 如果为 0(GPIO_ACTIVE_HIGH) 的话表示高电平有效, 如果为 1(GPIO_ACTIVE_LOW) 的话表示低电平有效。

示例代码 10.2.2.1 中的是 GPIO0 控制器节点, 当某个具体的引脚作为 GPIO 使用的时候还

需要进一步设置。正点原子 ATK-DLRK3568 开发板将 GPIO3_B6 用作 CSI1 摄像头的 RESET 引脚,GPIO3_B6 复用为 GPIO 功能,通过控制这个 GPIO 的高低电平就可以复位 CSI1 摄像头。但是,CSI1 摄像头驱动程序怎么知道 RESET 引脚连接的 GPIO3_B6 呢?这里肯定需要设备树来告诉驱动,在设备树中的 CSI1 摄像头节点下添加一个属性来描述摄像头的 RESET 引脚就行了,CSI1 摄像头驱动直接读取这个属性值就知道摄像头的 RESET 引脚使用的是哪个 GPIO 了。正点原子 ATK-DLRK3568 开发板的 CSI1 摄像头的 I2C 配置接口连接到 RK3568 的 I2C4 上。在 rk3568-atk-evb1-ddr4-v10.dtsi 中找到名为“i2c4”的节点,这个节点包含了所有连接到 I2C5 接口上的设备,如下所示:

示例代码 10.2.2.2 设备树中 I2C5 节点

```

585     imx415: imx415@1a {
586         status = "okay";
587         compatible = "sony,imx415";
588         reg = <0x1a>;
589         clocks = <&cru CLK_CIF_OUT>;
590         clock-names = "xvclk";
591         power-domains = <&power RK3568_PD_VI>;
592         pinctrl-names = "rockchip,camera_default";
593         pinctrl-0 = <&cif_clk>;
594         reset-gpios = <&gpio3 RK_PB6 GPIO_ACTIVE_LOW>;
595         power-gpios = <&gpio4 RK_PB4 GPIO_ACTIVE_HIGH>;
596         rockchip,camera-module-index = <0>;
597         rockchip,camera-module-facing = "back";
598         rockchip,camera-module-name = "CMK-OT1522-FG3";
599         rockchip,camera-module-lens-name = "CS-P1150-IRC-8M-FAU";
        .....
606     };
    
```

第 594 行,属性“reset-gpios”描述了 IMX415 这个摄像头的 RESET 引脚使用的哪个 IO。属性值一共有三个,我们来看一下这三个属性值的含义,“&gpio3”表示 RESET 引脚所使用的 IO 属于 GPIO3 组,“RK_PB6”表示 GPIO3 组的 PB6,通过这两个值 IMX415 摄像头驱动程序就知道 RESET 引脚使用了 GPIO3_B6 这个引脚。最后一个是“GPIO_ACTIVE_LOW”,Linux 内核在 include/linux/gpio/machine.h 文件中定义了枚举类型 gpio_lookup_flags,内容如下:

示例代码 10.2.2.3 gpio_lookup_flag 枚举类型

```

8  enum gpio_lookup_flags {
9      GPIO_ACTIVE_HIGH      = (0 << 0),
10     GPIO_ACTIVE_LOW       = (1 << 0),
11     GPIO_OPEN_DRAIN      = (1 << 1),
12     GPIO_OPEN_SOURCE     = (1 << 2),
13     GPIO_PERSISTENT      = (0 << 3),
14     GPIO_TRANSITORY      = (1 << 3),
15 };
    
```

10.2.3 gpio 子系统 API 函数

对于驱动开发人员, 设置好设备树以后就可以使用 `gpio` 子系统提供的 API 函数来操作指定的 GPIO, `gpio` 子系统向驱动开发人员屏蔽了具体的读写寄存器过程。这就是驱动分层与分离的好处, 大家各司其职, 做好自己的本职工作即可。`gpio` 子系统提供的常用的 API 函数有下面几个:

1、gpio_request 函数

`gpio_request` 函数用于申请一个 GPIO 管脚, 在使用一个 GPIO 之前一定要使用 `gpio_request` 进行申请, 函数原型如下:

```
int gpio_request(unsigned gpio, const char *label)
```

函数参数和返回值含义如下:

gpio: 要申请的 gpio 标号, 使用 `of_get_named_gpio` 函数从设备树获取指定 GPIO 属性信息, 此函数会返回这个 GPIO 的标号。

label: 给 gpio 设置个名字。

返回值: 0, 申请成功; 其他值, 申请失败。

2、gpio_free 函数

如果不使用某个 GPIO 了, 那么就可以调用 `gpio_free` 函数进行释放。函数原型如下:

```
void gpio_free(unsigned gpio)
```

函数参数和返回值含义如下:

gpio: 要释放的 gpio 标号。

返回值: 无。

3、gpio_direction_input 函数

此函数用于设置某个 GPIO 为输入, 函数原型如下所示:

```
int gpio_direction_input(unsigned gpio)
```

函数参数和返回值含义如下:

gpio: 要设置为输入的 GPIO 标号。

返回值: 0, 设置成功; 负值, 设置失败。

4、gpio_direction_output 函数

此函数用于设置某个 GPIO 为输出, 并且设置默认输出值, 函数原型如下:

```
int gpio_direction_output(unsigned gpio, int value)
```

函数参数和返回值含义如下:

gpio: 要设置为输出的 GPIO 标号。

value: GPIO 默认输出值。

返回值: 0, 设置成功; 负值, 设置失败。

5、gpio_get_value 函数

此函数用于获取某个 GPIO 的值(0 或 1), 函数原型如下:

```
int gpio_get_value(unsigned int gpio)
```

函数参数和返回值含义如下:

gpio: 要获取的 GPIO 标号。

返回值: 非负值, 得到的 GPIO 值; 负值, 获取失败。

6、gpio_set_value 函数

此函数用于设置某个 GPIO 的值，函数原型如下：

```
void gpio_set_value(unsigned int gpio, int value)
```

函数参数和返回值含义如下：

gpio: 要设置的 GPIO 标号。

value: 要设置的值。

返回值: 无

关于 gpio 子系统常用的 API 函数就讲这些，这些是我们用的最多的。

10.2.4 设备树中添加 gpio 节点模板

本节我们以正点原子 ATK-DLRK3568 开发板上的 LED 为例，学习一下如何创建 GPIO 节点。LED 连接到了 GPIO0_C0 引脚上，首先创建一个“led”设备节点。

1、创建 led 设备节点

在根节点“/”下创建 led 设备子节点，如下所示：

示例代码 10.2.4.1 led 设备节点

```
1 gpioled {
2     /* 节点内容 */
3 };
```

2、添加 GPIO 属性信息

在 gpioled 节点中添加 GPIO 属性信息，表明所使用的 GPIO 是哪个引脚，添加完成以后如下所示：

示例代码 10.2.4.2 向 gpioled 节点添加 gpio 属性

```
1 gpioled {
2     compatible = "alientek,led";
3     led-gpio = <&gpio0 RK_PC0 GPIO_ACTIVE_HIGH>;
4     status = "okay";
5 };
```

第 3 行，LED0 设备所使用的 gpio。

关于 pinctrl 子系统和 gpio 子系统就讲解到这里，接下来就使用 pinctrl 和 gpio 子系统来驱动 ATK-DLRK3568 开发板上的 LED 灯。

10.2.5 与 gpio 相关的 OF 函数

在示例代码 10.2.4.2 中，我们定义了一个名为“gpio”的属性，gpio 属性描述了 led 这个设备所使用的 GPIO。在驱动程序中需要读取 gpio 属性内容，Linux 内核提供了几个与 GPIO 有关的 OF 函数，常用的几个 OF 函数如下所示：

1、of_gpio_named_count 函数

of_gpio_named_count 函数用于获取设备树某个属性里面定义了几个 GPIO 信息，要注意的是空的 GPIO 信息也会被统计到，比如：

```
gpios = <0
        &gpio1 1 2
        0
        &gpio2 3 4>;
```

上述代码的“gpios”节点一共定义了 4 个 GPIO，但是有 2 个是空的，没有实际的含义。通过 of_gpio_named_count 函数统计出来的 GPIO 数量就是 4 个，此函数原型如下：

```
int of_gpio_named_count(struct device_node *np, const char *propname)
```

函数参数和返回值含义如下：

np: 设备节点。

propname: 要统计的 GPIO 属性。

返回值: 正值，统计到的 GPIO 数量；负值，失败。

2、of_gpio_count 函数

和 of_gpio_named_count 函数一样，但是不同的地方在于，此函数统计的是“gpios”这个属性的 GPIO 数量，而 of_gpio_named_count 函数可以统计任意属性的 GPIO 信息，函数原型如下所示：

```
int of_gpio_count(struct device_node *np)
```

函数参数和返回值含义如下：

np: 设备节点。

返回值: 正值，统计到的 GPIO 数量；负值，失败。

3、of_get_named_gpio 函数

此函数获取 GPIO 编号，因为 Linux 内核中关于 GPIO 的 API 函数都要使用 GPIO 编号，此函数会将设备树中类似<&gpio4 RK_PA1 GPIO_ACTIVE_LOW>的属性信息转换为对应的 GPIO 编号，此函数在驱动中使用很频繁！函数原型如下：

```
int of_get_named_gpio(struct device_node *np,
                    const char *propname,
                    int index)
```

函数参数和返回值含义如下：

np: 设备节点。

propname: 包含要获取 GPIO 信息的属性名。

index: GPIO 索引，因为一个属性里面可能包含多个 GPIO，此参数指定要获取哪个 GPIO 的编号，如果只有一个 GPIO 信息的话此参数为 0。

返回值: 正值，获取到的 GPIO 编号；负值，失败。

10.3 硬件原理图分析

本实验的硬件原理参考 6.2 小节即可。

10.4 实验程序编写

本实验对应的例程路径为：[开发板光盘](#)→[01、程序源码](#)→[Linux 驱动例程源码](#)→[05_gpioled](#)。

本章实验我们继续研究 LED 灯，在第九章实验中我们通过设备树向 dtsled.c 文件传递相应的寄存器物理地址，然后在驱动文件中配置寄存器。本章实验我们使用 gpio 子系统来完成 LED 灯驱动。

10.4.1 检查 IO 是否已经被使用

首先我们要检查一下 GPIO0_C0 对应的 GPIO 有没有被其他外设使用，如果这个 GPIO 已经被分配给了其他外设，那么我们驱动在申请这个 GPIO 就会失败，如图 10.4.1.1 所示：

```
root@ATK-DLRK356X:/lib/modules/4.19.232# modprobe gpioled
[ 43.393419] led-gpio num = 16
[ 43.393485] gpioled: Failed to request led-gpio ← GPIO申请失败
modprobe: ERROR: could not insert 'gpioled': Device or resource busy
root@ATK-DLRK356X:/lib/modules/4.19.232#
```

图 10.4.1.1 GPIO 申请失败

图 10.4.1.1 就是 GPIO 申请失败提示，因为当前开发板系统将 GPIO0_C0 这个 IO 分配给了内核自带的 LED 驱动做心跳灯了。所以需要先关闭 GPIO0_C0 作为心跳灯这个功能，也就是将 GPIO0_C0 对应的 GPIO 释放出来。打开 rk3568-evb.dtsi 文件，找到如下图 10.4.1.2 所示代码：

```
161
162     leds: leds {
163         compatible = "gpio-leds";
164         work_led: work {
165             gpios = <&gpio0 RK_PC0 GPIO_ACTIVE_HIGH>;
166             linux,default-trigger = "heartbeat";
167         };
168     };
```

图 10.4.1.2 GPIO0_C0 用作心跳灯

图 10.4.1.2 中可以看出，正点原子出厂系统默认将 GPIO0_C0 用作心跳灯了，所以系统在启动的时候就先将 GPIO0_C0 给了心跳灯，我们后面再申请肯定就会失败！

解决方法就是关闭心跳灯功能，也就是在图 10.4.1.2 中第 167 行添加 status 改为 disabled，也就是禁止 work 这个节点，那么禁止心跳灯功能。这样系统启动的时候就不会将 GPIO0_C0 分配给心跳灯，我们后面也就可以申请了，修改后如图 10.4.1.3 所示：

```
162     leds: leds {
163         compatible = "gpio-leds";
164         work_led: work {
165             gpios = <&gpio0 RK_PC0 GPIO_ACTIVE_HIGH>;
166             linux,default-trigger = "heartbeat";
167             status = "disabled"; ← 禁用led心跳灯
168         };
169     };
170
```

图 10.4.1.3 禁止心跳灯

图 10.4.1.3 中将 status 改为 disabled 就禁止了 led，也就是心跳灯功能，我们后面需要禁止哪个功能，只需要将其 status 属性改为 disabled 就可以了。

10.4.2 修改设备树文件

在 rk3568-atk-evb1-ddr4-v10.dtsi 文件的根节点“/”下创建 LED 灯节点，节点名为“gpioled”，节点内容如下：

示例代码 10.4.1.1 创建 LED 灯节点

```
1 gpioled {
2     compatible = "alientek,led";
3     led-gpio = <&gpio0 RK_PC0 GPIO_ACTIVE_HIGH>;
4     status = "okay";
5 };
```

第 3 行，led-gpio 属性指定了 LED 灯所使用的 GPIO，在这里就是 GPIO0 的 PC0，高电平有效。稍后编写驱动程序的时候会获取 led-gpio 属性的内容来得到 GPIO 编号，因为 gpio 子系统的 API 操作函数需要 GPIO 编号。

设备树编写完成以后使用“./build.sh kernel”命令重新编译并打包内核，然后将新编译出来

的 boot.img 烧写到开发板中。启动成功以后进入 “/proc/device-tree” 目录中查看 “gpioled” 节点是否存在，如果存在的话就说明设备树基本修改成功(具体还要驱动验证)，结果如图 10.4.2.1 所示：

```

eink@fdf00000                                rkvddec@fdf80200
ethernet@fe010000                            rkvinc@fdf40000
ethernet@fe2a0000                            rkvinc-opp-table
external-gmac0-clock                        rm500u-5g
external-gmac1-clock                        rng@fe388000
fiq-debugger                                rockchip-suspend
firmware                                     rockchip-system-monitor
gpioled ← gpioled子节点                 saradc@fe720000
gpio-regulator                              sata@fc000000
gpu@fde60000                                sata@fc400000
hdmi@fe0a0000                               sata@fc800000
hdmi-sound                                  scmi-shmem@10f000
i2c@fdd40000                                sdhci@fe310000
    
```

图 10.4.2.1 gpioled 子节点

10.4.3 LED 灯驱动程序编写

设备树准备好以后就可以编写驱动程序了，本章实验在第九章实验驱动文件 dtsled.c 的基础上修改而来。新建名为 “05_gpioled” 文件夹，然后在 05_gpioled 文件夹里面创建 vscode 工程，工作区命名为 “gpioled”。工程创建好以后新建 gpioled.c 文件，在 gpioled.c 里面输入如下内容：

```

示例代码 10.4.3.1 gpioled.c 驱动文件代码
1  #include <linux/types.h>
2  #include <linux/kernel.h>
3  #include <linux/delay.h>
4  #include <linux/ide.h>
5  #include <linux/init.h>
6  #include <linux/module.h>
7  #include <linux/errno.h>
8  #include <linux/gpio.h>
9  #include <linux/cdev.h>
10 #include <linux/device.h>
11 #include <linux/of.h>
12 #include <linux/of_address.h>
13 #include <linux/of_gpio.h>
14 // #include <asm/mach/map.h>
15 #include <asm/uaccess.h>
16 #include <asm/io.h>
17 /*****
18 Copyright © ALIENTEK Co., Ltd. 1998-2029. All rights reserved.
19 文件名      : gpioled.c
20 作者        : 正点原子
21 版本        : V1.0
22 描述        : LED 驱动文件。
23 其他        : 无
    
```

```

24 论坛          : www.openedv.com
25 日志          : 初版 v1.0 2022/12/07 正点原子团队创建
26 *****/
27 #define GPIOLED_CNT          1          /* 设备号个数 */
28 #define GPIOLED_NAME        "gpioled" /* 名字 */
29 #define LEDOFF               0          /* 关灯 */
30 #define LEDON                1          /* 开灯 */
31
32 /* gpioled 设备结构体 */
33 struct gpioled_dev{
34     dev_t devid;              /* 设备号 */
35     struct cdev cdev;        /* cdev */
36     struct class *class;     /* 类 */
37     struct device *device;   /* 设备 */
38     int major;               /* 主设备号 */
39     int minor;               /* 次设备号 */
40     struct device_node *nd;  /* 设备节点 */
41     int led_gpio;            /* led 所使用的 GPIO 编号 */
42 };
43
44 struct gpioled_dev gpioled; /* led 设备 */
45
46 /*
47  * @description : 打开设备
48  * @param - inode : 传递给驱动的 inode
49  * @param - filp : 设备文件, file 结构体有个叫做 private_data 的成员变量
50  *                一般在 open 的时候将 private_data 指向设备结构体。
51  * @return      : 0 成功;其他 失败
52  */
53 static int led_open(struct inode *inode, struct file *filp)
54 {
55     filp->private_data = &gpioled; /* 设置私有数据 */
56     return 0;
57 }
58
59 /*
60  * @description : 从设备读取数据
61  * @param - filp : 要打开的设备文件 (文件描述符)
62  * @param - buf : 返回给用户空间的数据缓冲区
63  * @param - cnt : 要读取的数据长度
64  * @param - offt : 相对于文件首地址的偏移
65  * @return      : 读取的字节数, 如果为负值, 表示读取失败
66  */
    
```

```

67 static ssize_t led_read(struct file *filp, char __user *buf,
                          size_t cnt, loff_t *offt)
68 {
69     return 0;
70 }
71
72 /*
73  * @description   : 向设备写数据
74  * @param - filp  : 设备文件, 表示打开的文件描述符
75  * @param - buf   : 要写给设备写入的数据
76  * @param - cnt   : 要写入的数据长度
77  * @param - offt  : 相对于文件首地址的偏移
78  * @return        : 写入的字节数, 如果为负值, 表示写入失败
79  */
80 static ssize_t led_write(struct file *filp, const char __user *buf,
                          size_t cnt, loff_t *offt)
81 {
82     int retvalue;
83     unsigned char databuf[1];
84     unsigned char ledstat;
85     struct gpioled_dev *dev = filp->private_data;
86
87     retvalue = copy_from_user(databuf, buf, cnt);
88     if(retvalue < 0) {
89         printk("kernel write failed!\r\n");
90         return -EFAULT;
91     }
92
93     ledstat = databuf[0];      /* 获取状态值 */
94
95     if(ledstat == LEDON) {
96         gpio_set_value(dev->led_gpio, 1); /* 打开 LED 灯 */
97     } else if(ledstat == LEDOFF) {
98         gpio_set_value(dev->led_gpio, 0); /* 关闭 LED 灯 */
99     }
100    return 0;
101 }
102
103 /*
104  * @description   : 关闭/释放设备
105  * @param - filp  : 要关闭的设备文件(文件描述符)
106  * @return        : 0 成功;其他 失败
107  */
    
```

```
108 static int led_release(struct inode *inode, struct file *filp)
109 {
110     return 0;
111 }
112
113 /* 设备操作函数 */
114 static struct file_operations gpioled_fops = {
115     .owner = THIS_MODULE,
116     .open = led_open,
117     .read = led_read,
118     .write = led_write,
119     .release = led_release,
120 };
121
122 /*
123  * @description   : 驱动出口函数
124  * @param         : 无
125  * @return        : 无
126  */
127 static int __init led_init(void)
128 {
129     int ret = 0;
130     const char *str;
131
132     /* 设置 LED 所使用的 GPIO */
133     /* 1、获取设备节点: gpioled */
134     gpioled.nd = of_find_node_by_path("/gpioled");
135     if(gpioled.nd == NULL) {
136         printk("gpioled node not find!\r\n");
137         return -EINVAL;
138     }
139
140     /* 2.读取 status 属性 */
141     ret = of_property_read_string(gpioled.nd, "status", &str);
142     if(ret < 0)
143         return -EINVAL;
144
145     if (strcmp(str, "okay"))
146         return -EINVAL;
147
148     /* 3、获取 compatible 属性值并进行匹配 */
149     ret = of_property_read_string(gpioled.nd, "compatible", &str);
150     if(ret < 0) {
```

```

151     printk("gpioled: Failed to get compatible property\n");
152     return -EINVAL;
153 }
154
155 if (strcmp(str, "alientek,led")) {
156     printk("gpioled: Compatible match failed\n");
157     return -EINVAL;
158 }
159
160 /* 4、获取设备树中的 gpio 属性，得到 LED 所使用的 LED 编号 */
161 gpioled.led_gpio = of_get_named_gpio(gpioled.nd, "led-gpio", 0);
162 if(gpioled.led_gpio < 0) {
163     printk("can't get led-gpio");
164     return -EINVAL;
165 }
166 printk("led-gpio num = %d\r\n", gpioled.led_gpio);
167
168 /* 5.向 gpio 子系统申请使用 GPIO */
169 ret = gpio_request(gpioled.led_gpio, "LED-GPIO");
170 if (ret) {
171     printk(KERN_ERR "gpioled: Failed to request led-gpio\n");
172     return ret;
173 }
174
175 /* 6、设置 GPIO 为输出，并且输出低电平，默认关闭 LED 灯 */
176 ret = gpio_direction_output(gpioled.led_gpio, 0);
177 if(ret < 0) {
178     printk("can't set gpio!\r\n");
179 }
180
181 /* 注册字符设备驱动 */
182 /* 1、创建设备号 */
183 if (gpioled.major) { /* 定义了设备号 */
184     gpioled.devid = MKDEV(gpioled.major, 0);
185     ret = register_chrdev_region(gpioled.devid, GPIOLED_CNT,
186                                 GPIOLED_NAME);
187
188     if(ret < 0) {
189         pr_err("cannot register %s char driver [ret=%d]\n",
190               GPIOLED_NAME, GPIOLED_CNT);
191         goto free_gpio;
192     }
193 } else { /* 没有定义设备号 */
194     ret = alloc_chrdev_region(&gpioled.devid, 0, GPIOLED_CNT,

```

```

GPIOLED_NAME); /* 申请设备号 */
192     if(ret < 0) {
193         pr_err("%s Couldn't alloc_chrdev_region, ret=%d\r\n",
                GPIOLED_NAME, ret);
194         goto free_gpio;
195     }
196     gpioled.major = MAJOR(gpioled.devid); /* 获取分配号的主设备号 */
197     gpioled.minor = MINOR(gpioled.devid); /* 获取分配号的次设备号 */
198 }
199 printk("gpioled major=%d,minor=%d\r\n",gpioled.major,
        gpioled.minor);
200
201 /* 2、初始化 cdev */
202 gpioled.cdev.owner = THIS_MODULE;
203 cdev_init(&gpioled.cdev, &gpioled_fops);
204
205 /* 3、添加一个 cdev */
206 cdev_add(&gpioled.cdev, gpioled.devid, GPIOLED_CNT);
207 if(ret < 0)
208     goto del_unregister;
209
210 /* 4、创建类 */
211 gpioled.class = class_create(THIS_MODULE, GPIOLED_NAME);
212 if (IS_ERR(gpioled.class)) {
213     goto del_cdev;
214 }
215
216 /* 5、创建设备 */
217 gpioled.device = device_create(gpioled.class, NULL,
                                gpioled.devid, NULL, GPIOLED_NAME);
218 if (IS_ERR(gpioled.device)) {
219     goto destroy_class;
220 }
221 return 0;
222
223 destroy_class:
224     class_destroy(gpioled.class);
225 del_cdev:
226     cdev_del(&gpioled.cdev);
227 del_unregister:
228     unregister_chrdev_region(gpioled.devid, GPIOLED_CNT);
229 free_gpio:
230     gpio_free(gpioled.led_gpio);
    
```

```

231     return -EIO;
232 }
233
234 /*
235  * @description   : 驱动出口函数
236  * @param         : 无
237  * @return        : 无
238  */
239 static void __exit led_exit(void)
240 {
241     /* 注销字符设备驱动 */
242     cdev_del(&gpioled.cdev); /* 删除 cdev */
243     unregister_chrdev_region(gpioled.devid, GPIOLED_CNT);
244     device_destroy(gpioled.class, gpioled.devid); /* 注销设备 */
245     class_destroy(gpioled.class); /* 注销类 */
246     gpio_free(gpioled.led_gpio); /* 释放 GPIO */
247 }
248
249 module_init(led_init);
250 module_exit(led_exit);
251 MODULE_LICENSE("GPL");
252 MODULE_AUTHOR("ALIEN TEK");
253 MODULE_INFO(intree, "Y");
    
```

第 41 行, 在设备结构体 `gpioled_dev` 中加入 `led_gpio` 这个成员变量, 此成员变量保存 LED 等所使用的 GPIO 编号。

第 55 行, 将设备结构体变量 `gpioled` 设置为 `filp` 的私有数据 `private_data`。

第 85 行, 通过读取 `filp` 的 `private_data` 成员变量来得到设备结构体变量, 也就是 `gpioled`。这种将设备结构体设置为 `filp` 私有数据的方法在 Linux 内核驱动里面非常常见。

第 96、98 行, 直接调用 `gpio_set_value` 函数来向 GPIO 写入数据, 实现开/关 LED 的效果。不需要我们直接操作相应的寄存器。

第 134 行, 获取节点 `"/gpioled"`。

第 141~146 行, 获取 `"status"` 属性的值, 判断属性是否 `"okay"`。

第 149~158 行, 获取 `compatible` 属性值并进行匹配。

第 161 行, 通过函数 `of_get_named_gpio` 函数获取 LED 所使用的 LED 编号。相当于将 `gpioled` 节点中的 `"led-gpio"` 属性值转换为对应的 LED 编号。

第 169 行, 通过函数 `gpio_request` 向 GPIO 子系统申请使用 `GPIO0_C0`。

第 176 行, 调用函数 `gpio_direction_output` 设置 `GPIO0_C0` 这个 GPIO 为输出, 并且默认低电平, 这样默认就会关闭 LED 灯。

可以看出 `gpioled.c` 文件中的内容和第九章的 `dtsled.c` 差不多, 只是取消掉了配置寄存器的过程, 改为使用 Linux 内核提供的 API 函数。在 GPIO 操作上更加的规范化, 符合 Linux 代码框架, 而且也简化了 GPIO 驱动开发的难度, 以后我们所有例程用到 GPIO 的地方都采用此方法。

10.4.4 编写测试 APP

本章直接使用第九章的测试 APP，将上一章的 ledApp.c 文件复制到本章实验工程下即可。

10.5 运行测试

10.5.1 编译驱动程序和测试 APP

1、编译驱动程序

编写 Makefile 文件，本章实验的 Makefile 文件和第五章实验基本一样，只是将 obj-m 变量的值改为 gpioled.o，Makefile 内容如下所示：

示例代码 10.5.1.1 Makefile 文件

```
1 KERNELDIR := /home/alientek/rk3568_linux_sdk/kernel
.....
4 obj-m := gpioled.o
.....
11 clean:
12 $(MAKE) -C $(KERNELDIR) M=$(CURRENT_PATH) clean
```

第 4 行，设置 obj-m 变量的值为 gpioled.o。

输入如下命令编译出驱动模块文件：

```
make ARCH=arm64 //ARCH=arm64 必须指定，否则编译会失败
```

编译成功以后就会生成一个名为“gpioled.ko”的驱动模块文件。

2、编译测试 APP

输入如下命令编译测试 ledApp.c 这个测试程序：

```
/opt/atk-dlrk356x-toolchain/bin/aarch64-buildroot-linux-gnu-gcc ledApp.c -o ledApp
```

编译成功以后就会生成 ledApp 这个应用程序。

10.5.2 运行测试

由于在本实验中，我们直接通过修改设备树的方式永久的关闭了 LED0 心跳灯功能，所以就不需要再输入命令关闭了。

在 Ubuntu 中将上一小节编译出来的 gpioled.ko 和 ledApp 这两个文件通过 adb 命令发送到开发板的/lib/modules/4.19.232 目录下，命令如下：

```
adb push gpioled.ko ledApp /lib/modules/4.19.232
```

发送成功以后进入到开发板目录 lib/modules/4.19.232 中，输入如下命令加载 dtsled.ko 驱动模块：

```
depmod //第一次加载驱动的时候需要运行此命令
modprobe gpioled //加载驱动
```

驱动加载成功以后会在终端中输出一些信息，如图 10.5.2.1 所示：

```
root@ATK-DLRK356X:/lib/modules/4.19.232# modprobe gpioled
[139917.420269] led-gpio num = 16
[139917.420349] gpioled major=236,minor=0
root@ATK-DLRK356X:/lib/modules/4.19.232#
```

图 10.5.2.1 驱动加载成功以后输出的信息

从图 10.5.2.1 可以看出，gpioled 这个节点找到了，并且 GPIO0_C0 这个 GPIO 的编号为 124。

驱动加载成功以后就可以使用 `ledApp` 软件来测试驱动是否工作正常，输入如下命令打开 LED 灯：

```
./ledApp /dev/gpioled 1 //打开 LED 灯
```

输入上述命令以后观察开发板上的红色 LED 灯是否点亮，如果点亮的话说明驱动工作正常。在输入如下命令关闭 LED 灯：

```
./ledApp /dev/gpioled 0 //关闭 LED 灯
```

输入上述命令以后观察开发板上的红色 LED 灯是否熄灭。如果要卸载驱动的话输入如下命令即可：

```
rmmod gpioled.ko
```

第十一章 Linux 并发与竞争

Linux 是一个多任务操作系统,肯定会存在多个任务共同操作同一段内存或者设备的情况,多个任务甚至中断都能访问的资源叫做共享资源,就和共享单车一样。在驱动开发中要注意对共享资源的保护,也就是要处理对共享资源的并发访问。比如共享单车,大家按照谁扫谁骑走的原则来共用这个单车,如果没有这个并发访问共享单车的原则存在,只怕到时候为了一辆单车要打起来了。在 Linux 驱动编写过程中对于并发控制的管理非常重要,本章我们就来学习一下如何在 Linux 驱动中处理并发。

11.1 并发与竞争

1、并发与竞争简介

并发就是多个“用户”同时访问同一个共享资源，比如你们公司有一台打印机，你们公司的所有人都可以使用。现在小李和小王要同时使用这一台打印机，都要打印一份文件。小李要打印的文件内容如下：

示例代码 11.1.1 小李要打印的内容

```
我叫小李  
电话：123456  
工号：16
```

小王要打印的内容如下：

示例代码 11.1.2 小王要打印的内容

```
我叫小王  
电话：678910  
工号：20
```

这两份文档肯定是各自打印出来的，不能相互影响。当两个人同时打印的时候，如果打印机不做处理，可能会出现小李的文档打印了一行，然后开始打印小王的文档，这样打印出来的文档就错乱了，可能会出现如下的错误文档内容：

示例代码 11.1.3 小王打印出来的错误文档

```
我叫小王  
电话：123456  
工号：20
```

可以看出，小王打印出来的文档中电话号码错误了，变成小李的了，这是绝对不允许的。如果有多人同时向打印机发送了多份文档，打印机必须保证一次只能打印一份文档，只有打印完成以后才能打印其他的文档。

Linux 系统是个多任务操作系统，会存在多个任务同时访问同一片内存区域，这些任务可能会相互覆盖这段内存中的数据，造成内存数据混乱。针对这个问题必须要做处理，严重的话可能会导致系统崩溃。现在的 Linux 系统并发产生的原因很复杂，总结一下有下面几个主要原因：

- ①、多线程并发访问，Linux 是多任务(线程)的系统，所以多线程访问是最基本的原因。
- ②、抢占式并发访问，从 2.6 版本内核开始，Linux 内核支持抢占，也就是说调度程序可以在任意时刻抢占正在运行的线程，从而运行其他的线程。
- ③、中断程序并发访问，这个无需多说，学过 STM32 的同学应该知道，硬件中断的权利可是很大的。
- ④、SMP(多核)核间并发访问，现在 ARM 架构的多核 SOC 很常见，多核 CPU 存在核间并发访问。

并发访问带来的问题就是竞争，学过 FreeRTOS 或 UCOS 这类 RTOS 的同学应该知道临界区这个概念，所谓的临界区就是共享数据段，对于临界区必须保证一次只有一个线程访问，也就是要保证临界区是原子访问的，注意这里的“原子”不是正点原子的“原子”。我们都知道，原子是化学反应不可再分的基本微粒，这里的原子访问就表示这一个访问是一个步骤，不能再进行拆分。如果多个线程同时操作临界区就表示存在竞争，我们在编写驱动的时候一定要注意避免并发和防止竞争访问。很多 Linux 驱动初学者往往不注意这一点，在驱动程序中埋下了隐

患，这类问题往往又很不容易查找，导致驱动调试难度加大、费时费力。所以我们一般在编写驱动的时候就要考虑到并发与竞争，而不是驱动都编写完了然后再处理并发与竞争。

2、保护内容是什么

前面一直说要防止并发访问共享资源，换句话说就是要保护共享资源，防止进行并发访问。那么问题来了，什么是共享资源？现实生活中的公共电话、共享单车这些是共享资源，我们很容易理解，那么在程序中什么是共享资源？也就是保护的内容是什么？我们保护的并不是代码，而是数据！某个线程的局部变量不需要保护，我们要保护的是多个线程都会访问的共享数据。一个整形的全局变量 `a` 是数据，一份要打印的文档也是数据，虽然我们知道了要对共享数据进行保护，那么怎么判断哪些共享数据要保护呢？找到要保护的数据才是重点，而这个也是难点，因为驱动程序各不相同，那么数据也千变万化，一般像全局变量，设备结构体这些肯定是要保护的，至于其他的数据就要根据实际的驱动程序而定了。

当我们发现驱动程序中存在并发和竞争的时候一定要处理掉，接下来我们依次来学习一下 Linux 内核提供的几种并发和竞争的处理方法。

11.2 原子操作

11.2.1 原子操作简介

首先看一下原子操作，原子操作就是指不能再进一步分割的操作，一般原子操作用于变量或者位操作。假如现在要对无符号整形变量 `a` 赋值，值为 3，对于 C 语言来讲很简单，直接就是：

```
a = 3
```

但是 C 语言要先编译为成汇编指令，ARM 架构不支持直接对寄存器(内存)进行读写操作，要借助寄存器 `R0`、`R1` 等来完成赋值操作。假设变量 `a` 的地址为 `0X3000000`，“`a=3`”这一行 C 语言可能会被编译为如下所示的汇编代码：

```

示例代码 11.2.1.1 汇编示例代码
1 ldr r0, =0x30000000 /* 变量 a 地址 */
2 ldr r1, = 3 /* 要写入的值 */
3 str r1, [r0] /* 将 3 写入到 a 变量中 */
    
```

示例代码 11.2.1.1 只是一个简单的举例说明，实际的结果要比示例代码复杂的多。从上述代码可以看出，C 语言里面简简单单的一句“`a=3`”，编译成汇编文件以后变成了 3 句，那么程序在执行的时候肯定是按照示例代码 11.2.1.1 中的汇编语句一条一条的执行。假设现在线程 A 要向 `a` 变量写入 10 这个值，而线程 B 也要向 `a` 变量写入 20 这个值，我们理想中的执行顺序如图 11.2.1.1 所示：

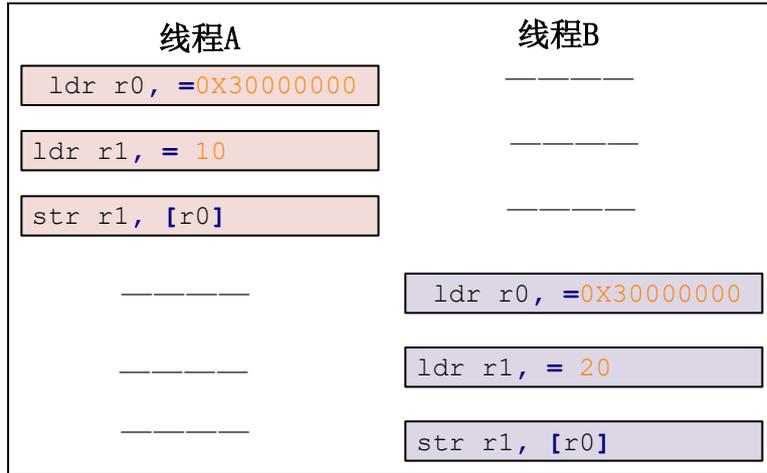


图 11.2.1.1 理想的执行流程

按照图 11.2.1.1 所示的流程，确实可以实现线程 A 将 a 变量设置为 10，线程 B 将 a 变量设置为 20。但是实际上的执行流程可能如图 11.2.1.2 所示：

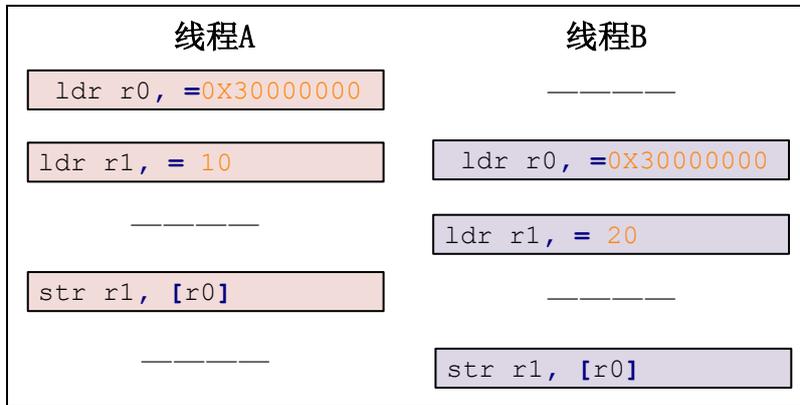


图 11.2.1.2 可能的执行流程

按照图 11.2.1.2 所示的流程，线程 A 最终将变量 a 设置为了 20，而并不是要求的 10！线程 B 没有问题。这就是一个最简单的设置变量值的并发与竞争的例子，要解决这个问题就要保证示例代码 11.2.1.1 中的三行汇编指令作为一个整体运行，也就是作为一个原子存在。Linux 内核提供了一组原子操作 API 函数来完成此功能，Linux 内核提供了两组原子操作 API 函数，一组是对整形变量进行操作的，一组是对位进行操作的，我们接下来看一下这些 API 函数。

11.2.2 原子整形操作 API 函数

Linux 内核定义了叫做 atomic_t 的结构体来完成整形数据的原子操作，在使用中用原子变量来代替整形变量，此结构体定义在 include/linux/types.h 文件中，定义如下：

示例代码 11.2.2.1 atomic_t 结构体

```
171 typedef struct {
172     int counter;
173 } atomic_t;
```

如果要使用原子操作 API 函数，首先要先定义一个 atomic_t 的变量，如下所示：

```
atomic_t a; //定义 a
```

也可以在定义原子变量的时候给原子变量赋初值，如下所示：

```
atomic_t b = ATOMIC_INIT(0); //定义原子变量 b 并赋初值为 0
```

可以通过宏 `ATOMIC_INIT` 向原子变量赋初值。

原子变量有了，接下来就是对原子变量进行操作，比如读、写、增加、减少等等，Linux 内核提供了大量的原子操作 API 函数，如表 11.2.2.1 所示：

函数	描述
<code>ATOMIC_INIT(int i)</code>	定义原子变量的时候对其初始化。
<code>int atomic_read(atomic_t *v)</code>	读取 <code>v</code> 的值，并且返回。
<code>void atomic_set(atomic_t *v, int i)</code>	向 <code>v</code> 写入 <code>i</code> 值。
<code>void atomic_add(int i, atomic_t *v)</code>	给 <code>v</code> 加上 <code>i</code> 值。
<code>void atomic_sub(int i, atomic_t *v)</code>	从 <code>v</code> 减去 <code>i</code> 值。
<code>void atomic_inc(atomic_t *v)</code>	给 <code>v</code> 加 1，也就是自增。
<code>void atomic_dec(atomic_t *v)</code>	从 <code>v</code> 减 1，也就是自减
<code>int atomic_dec_return(atomic_t *v)</code>	从 <code>v</code> 减 1，并且返回 <code>v</code> 的值。
<code>int atomic_inc_return(atomic_t *v)</code>	给 <code>v</code> 加 1，并且返回 <code>v</code> 的值。
<code>int atomic_sub_and_test(int i, atomic_t *v)</code>	从 <code>v</code> 减 <code>i</code> ，如果结果为 0 就返回真，否则返回假
<code>int atomic_dec_and_test(atomic_t *v)</code>	从 <code>v</code> 减 1，如果结果为 0 就返回真，否则返回假
<code>int atomic_inc_and_test(atomic_t *v)</code>	给 <code>v</code> 加 1，如果结果为 0 就返回真，否则返回假
<code>int atomic_add_negative(int i, atomic_t *v)</code>	给 <code>v</code> 加 <code>i</code> ，如果结果为负就返回真，否则返回假

表 11.2.2.1 原子整形操作 API 函数表

如果使用 64 位的 SOC 的话，就要用到 64 位的原子变量，Linux 内核也定义了 64 位原子结构体，如下所示：

示例代码 11.2.2.2 `atomic64_t` 结构体

```
176 typedef struct {
177     s64 counter;
178 } atomic64_t;
```

可以看出，`counter` 变为了 `s64` 类型，`s64` 本质是 `long long` 类型，相应的也提供了 64 位原子变量的操作 API 函数，这里我们就不详细讲解了，和表 11.2.1.1 中的 API 函数用法一样，只是将“`atomic_`”前缀换为“`atomic64_`”，将 `int` 换为 `long long`。如果使用的是 64 位的 SOC，那么就要使用 64 位的原子操作函数。Cortex-A7 是 32 位的架构，所以本书中只使用表 11.2.2.1 中的 32 位原子操作函数。原子变量和相应的 API 函数使用起来很简单，参考如下示例：

示例代码 11.2.2.2 原子变量和 API 函数使用

```
atomic_t v = ATOMIC_INIT(0); /* 定义并初始化原子变零 v=0 */
atomic_set(&v, 10); /* 设置 v=10 */
atomic_read(&v); /* 读取 v 的值，肯定是 10 */
atomic_inc(&v); /* v 的值加 1，v=11 */
```

11.2.3 原子位操作 API 函数

位操作也是很常用的操作，Linux 内核也提供了一系列的原子位操作 API 函数，只不过原子位操作不像原子整形变量那样有个 `atomic_t` 的数据结构，原子位操作是直接对内存进行操作，API 函数如表 11.2.3.1 所示：

函数	描述
<code>void set_bit(int nr, void *p)</code>	将 <code>p</code> 地址的第 <code>nr</code> 位置 1。

<code>void clear_bit(int nr, void *p)</code>	将 p 地址的第 nr 位清零。
<code>void change_bit(int nr, void *p)</code>	将 p 地址的第 nr 位进行翻转。
<code>int test_bit(int nr, void *p)</code>	获取 p 地址的第 nr 位的值。
<code>int test_and_set_bit(int nr, void *p)</code>	将 p 地址的第 nr 位置 1, 并且返回 nr 位原来的值。
<code>int test_and_clear_bit(int nr, void *p)</code>	将 p 地址的第 nr 位清零, 并且返回 nr 位原来的值。
<code>int test_and_change_bit(int nr, void *p)</code>	将 p 地址的第 nr 位翻转, 并且返回 nr 位原来的值。

表 11.2.3.1 原子位操作函数表

11.3 自旋锁

11.3.1 自旋锁简介

原子操作只能对整形变量或者位进行保护, 但是, 在实际的使用环境中怎么可能只有整形变量或位这么简单的临界区。举个最简单的例子, 设备结构体变量就不是整型变量, 我们对于结构体中成员变量的操作也要保证原子性, 在线程 A 对结构体变量使用期间, 应该禁止其他的线程来访问此结构体变量, 这些工作原子操作都不能胜任, 需要本节要讲的锁机制, 在 Linux 内核中就是自旋锁。

当一个线程要访问某个共享资源的时候首先要先获取相应的锁, 锁只能被一个线程持有, 只要此线程不释放持有的锁, 那么其他的线程就不能获取此锁。对于自旋锁而言, 如果自旋锁正在被线程 A 持有, 线程 B 想要获取自旋锁, 那么线程 B 就会处于忙循环-旋转-等待状态, 线程 B 不会进入休眠状态或者说去做其他的处理, 而是会一直傻傻的在那里“转圈圈”的等待锁可用。比如现在有个公用电话亭, 一次肯定只能进去一个人打电话, 现在电话亭里面有人正在打电话, 相当于获得了自旋锁。此时你到了电话亭门口, 因为里面有人, 所以你不能进去打电话, 相当于没有获取自旋锁, 这个时候你肯定是站在原地等待, 你可能因为无聊的等待而转圈圈消遣时光, 反正就是哪里也不能去, 要一直等到里面的人打完电话出来。终于, 里面的人打完电话出来了, 相当于释放了自旋锁, 这个时候你就可以使用电话亭打电话了, 相当于获取到了自旋锁。

自旋锁的“自旋”也就是“原地打转”的意思, “原地打转”的目的是为了等待自旋锁可以用, 可以访问共享资源。把自旋锁比作一个变量 a, 变量 a=1 的时候表示共享资源可用, 当 a=0 的时候表示共享资源不可用。现在线程 A 要访问共享资源, 发现 a=0(自旋锁被其他线程持有), 那么线程 A 就会不断的查询 a 的值, 直到 a=1。从这里我们可以看到自旋锁的一个缺点: 那就等待自旋锁的线程会一直处于自旋状态, 这样会浪费处理器时间, 降低系统性能, 所以自旋锁的持有时间不能太长。自旋锁适用于短时期的轻量级加锁, 如果遇到需要长时间持有锁的场景那就需要换其他的方法了, 这个我们后面会讲解。

Linux 内核使用结构体 `spinlock_t` 表示自旋锁, 结构体定义如下所示:

示例代码 11.3.1.1 spinlock_t 结构体

```

61 typedef struct spinlock {
62     union {
63         struct raw_spinlock rlock;
64
65 #ifdef CONFIG_DEBUG_LOCK_ALLOC
66 # define LOCK_PADSIZE (offsetof(struct raw_spinlock, dep_map))
67         struct {
68             u8 __padding[LOCK_PADSIZE];

```

```

69         struct lockdep_map dep_map;
70     };
71 #endif
72     };
73 } spinlock_t;
    
```

在使用自旋锁之前，肯定要先定义一个自旋锁变量，定义方法如下所示：

```
spinlock_t lock; //定义自旋锁
```

定义好自旋锁变量以后就可以使用相应的 API 函数来操作自旋锁。

11.3.2 自旋锁 API 函数

最基本的自旋锁 API 函数如表 11.3.2.1 所示：

函数	描述
DEFINE_SPINLOCK(spinlock_t lock)	定义并初始化一个自旋变量。
int spin_lock_init(spinlock_t *lock)	初始化自旋锁。
void spin_lock(spinlock_t *lock)	获取指定的自旋锁，也叫做加锁。
void spin_unlock(spinlock_t *lock)	释放指定的自旋锁。
int spin_trylock(spinlock_t *lock)	尝试获取指定的自旋锁，如果没有获取到就返回 0
int spin_is_locked(spinlock_t *lock)	检查指定的自旋锁是否被获取，如果没有被获取就返回非 0，否则返回 0。

表 11.3.2.1 自旋锁基本 API 函数表

表 11.3.2.1 中的自旋锁 API 函数适用于 SMP 或支持抢占的单 CPU 下线程之间的并发访问，也就是用于线程与线程之间，被自旋锁保护的临界区一定不能调用任何能够引起睡眠和阻塞的 API 函数，否则的话可能会导致死锁现象的发生。自旋锁会自动禁止抢占，也就是说当线程 A 得到锁以后会暂时禁止内核抢占。如果线程 A 在持有锁期间进入了休眠状态，那么线程 A 会自动放弃 CPU 使用权。线程 B 开始运行，线程 B 也想要获取锁，但是此时锁被 A 线程持有，而且内核抢占还被禁止了！线程 B 无法被调度出去，那么线程 A 就无法运行，锁也就无法释放，好了，死锁发生了！

表 11.3.2.1 中的 API 函数用于线程之间的并发访问，如果此时中断也要插一脚，中断也想访问共享资源，那该怎么办呢？首先可以肯定的是，中断里面可以使用自旋锁，但是在中断里面使用自旋锁的时候，在获取锁之前一定要先禁止本地中断(也就是本 CPU 中断，对于多核 SOC 来说会有多个 CPU 核)，否则可能导致锁死现象的发生，如图 11.3.2.1 所示：

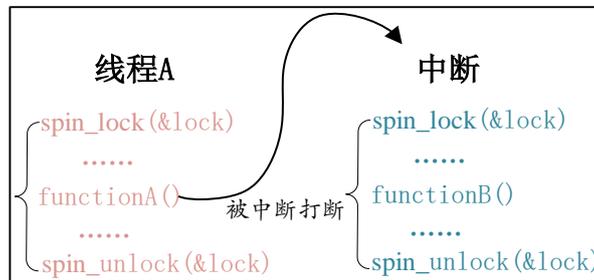


图 11.3.2.1 中断打断线程

在图 11.3.2.1 中，线程 A 先运行，并且获取到了 lock 这个锁，当线程 A 运行 functionA 函数的时候中断发生了，中断抢走了 CPU 使用权。右边的中断服务函数也要获取 lock 这个锁，但是这个锁被线程 A 占有着，中断就会一直自旋，等待锁有效。但是在中断服务函数执行完之

前, 线程 A 是不可能执行的, 线程 A 说“你先放手”, 中断说“你先放手”, 场面就这么僵持着, 死锁发生!

最好的解决方法就是获取锁之前关闭本地中断, Linux 内核提供了相应的 API 函数, 如表 11.3.2.2 所示:

函数	描述
<code>void spin_lock_irq(spinlock_t *lock)</code>	禁止本地中断, 并获取自旋锁。
<code>void spin_unlock_irq(spinlock_t *lock)</code>	激活本地中断, 并释放自旋锁。
<code>void spin_lock_irqsave(spinlock_t *lock, unsigned long flags)</code>	保存中断状态, 禁止本地中断, 并获取自旋锁。
<code>void spin_unlock_irqrestore(spinlock_t *lock, unsigned long flags)</code>	将中断状态恢复到以前的状态, 并且激活本地中断, 释放自旋锁。

表 11.3.2.2 线程与中断并发访问处理 API 函数

使用 `spin_lock_irq/spin_unlock_irq` 的时候需要用户能够确定加锁之前的中断状态, 但实际上内核很庞大, 运行也是“千变万化”, 我们是很难确定某个时刻的中断状态, 因此不推荐使用 `spin_lock_irq/spin_unlock_irq`。建议使用 `spin_lock_irqsave/spin_unlock_irqrestore`, 因为这一组函数会保存中断状态, 在释放锁的时候会恢复中断状态。一般在线程中使用 `spin_lock_irqsave/spin_unlock_irqrestore`, 在中断中使用 `spin_lock/spin_unlock`, 示例代码如下所示:

示例代码 11.3.2.1 自旋锁使用示例

```

1  DEFINE_SPINLOCK(lock)                /* 定义并初始化一个锁 */
2
3  /* 线程 A */
4  void functionA () {
5      unsigned long flags;              /* 中断状态 */
6      spin_lock_irqsave(&lock, flags)    /* 获取锁 */
7      /* 临界区 */
8      spin_unlock_irqrestore(&lock, flags) /* 释放锁 */
9  }
10
11 /* 中断服务函数 */
12 void irq() {
13     spin_lock(&lock)                   /* 获取锁 */
14     /* 临界区 */
15     spin_unlock(&lock)                  /* 释放锁 */
16 }
    
```

下半部(BH)也会竞争共享资源, 有些资料也会将下半部叫做底半部。关于下半部后面的章节会讲解, 如果要在下半部里面使用自旋锁, 可以使用表 11.3.2.3 中的 API 函数:

函数	描述
<code>void spin_lock_bh(spinlock_t *lock)</code>	关闭下半部, 并获取自旋锁。
<code>void spin_unlock_bh(spinlock_t *lock)</code>	打开下半部, 并释放自旋锁。

表 11.3.2.3 下半部竞争处理函数

11.3.3 其他类型的锁

在自旋锁的基础上还衍生出了其他特定场合使用的锁，这些锁在驱动中其实用的不多，更多的是在 Linux 内核中使用，本节我们简单来了解一下这些衍生出来的锁。

1、读写自旋锁

现在有个学生信息表，此表存放着学生的年龄、家庭住址、班级等信息，此表可以随时被修改和读取。此表肯定是数据，那么必须要对其进行保护，如果我们现在使用自旋锁对其进行保护。每次只能一个读操作或者写操作，但是，实际上此表是可以并发读取的。只需要保证在修改此表的时候没人读取，或者在其他人的时候没有人修改此表就行了。也就是此表的读和写不能同时进行，但是可以多人并发的读取此表。像这样，当某个数据结构符合读/写或生产者/消费者模型的时候就可以使用读写自旋锁。

读写自旋锁为读和写操作提供了不同的锁，一次只能允许一个写操作，也就是只能一个线程持有写锁，而且不能进行读操作。但是当没有写操作的时候允许一个或多个线程持有读锁，可以进行并发的读操作。Linux 内核使用 `rwlock_t` 结构体表示读写锁，结构体定义如下(删除了条件编译):

示例代码 11.3.3.1 `rwlock_t` 结构体

```
typedef struct {
    arch_rwlock_t raw_lock;
} rwlock_t;
```

读写锁操作 API 函数分为两部分，一个是给读使用的，一个是给写使用的，这些 API 函数如表 11.3.3.1 所示:

函数	描述
<code>DEFINE_RWLOCK(rwlock_t lock)</code>	定义并初始化读写锁
<code>void rwlock_init(rwlock_t *lock)</code>	初始化读写锁。
读锁	
<code>void read_lock(rwlock_t *lock)</code>	获取读锁。
<code>void read_unlock(rwlock_t *lock)</code>	释放读锁。
<code>void read_lock_irq(rwlock_t *lock)</code>	禁止本地中断，并且获取读锁。
<code>void read_unlock_irq(rwlock_t *lock)</code>	打开本地中断，并且释放读锁。
<code>void read_lock_irqsave(rwlock_t *lock, unsigned long flags)</code>	保存中断状态，禁止本地中断，并获取读锁。
<code>void read_unlock_irqrestore(rwlock_t *lock, unsigned long flags)</code>	将中断状态恢复到以前的状态，并且激活本地中断，释放读锁。
<code>void read_lock_bh(rwlock_t *lock)</code>	关闭下半部，并获取读锁。
<code>void read_unlock_bh(rwlock_t *lock)</code>	打开下半部，并释放读锁。
写锁	
<code>void write_lock(rwlock_t *lock)</code>	获取写锁。
<code>void write_unlock(rwlock_t *lock)</code>	释放写锁。
<code>void write_lock_irq(rwlock_t *lock)</code>	禁止本地中断，并且获取写锁。
<code>void write_unlock_irq(rwlock_t *lock)</code>	打开本地中断，并且释放写锁。
<code>void write_lock_irqsave(rwlock_t *lock, unsigned long flags)</code>	保存中断状态，禁止本地中断，并获取写锁。

<code>void write_unlock_irqrestore(rwlock_t *lock, unsigned long flags)</code>	将中断状态恢复到以前的状态，并且激活本地中断，释放读锁。
<code>void write_lock_bh(rwlock_t *lock)</code>	关闭下半部，并获取读锁。
<code>void write_unlock_bh(rwlock_t *lock)</code>	打开下半部，并释放读锁。

表 11.3.3.1 读写锁 API 函数

2、顺序锁

顺序锁在读写锁的基础上衍生而来的，使用读写锁的时候读操作和写操作不能同时进行。使用顺序锁的话可以允许在写的时候进行读操作，也就是实现同时读写，但是不允许同时进行并发的写操作。虽然顺序锁的读和写操作可以同时进行，但是如果在读的过程中发生了写操作，最好重新进行读取，保证数据完整性。顺序锁保护的资源不能是指针，因为如果在写操作的时候可能会导致指针无效，而这个时候恰巧有读操作访问指针的话就可能发生意外发生，比如读取野指针导致系统崩溃。Linux 内核使用 `seqlock_t` 结构体表示顺序锁，结构体定义如下：

 示例代码 11.3.3.2 `seqlock_t` 结构体

```

404 typedef struct {
405     struct seqcount seqcount;
406     spinlock_t lock;
407 } seqlock_t;
    
```

关于顺序锁的 API 函数如表 11.3.3.2 所示：

函数	描述
<code>DEFINE_SEQLOCK(seqlock_t sl)</code>	定义并初始化顺序锁
<code>void seqlock_ini(seqlock_t *sl)</code>	初始化顺序锁。
顺序锁写操作	
<code>void write_seqlock(seqlock_t *sl)</code>	获取写顺序锁。
<code>void write_sequnlock(seqlock_t *sl)</code>	释放写顺序锁。
<code>void write_seqlock_irq(seqlock_t *sl)</code>	禁止本地中断，并且获取写顺序锁
<code>void write_sequnlock_irq(seqlock_t *sl)</code>	打开本地中断，并且释放写顺序锁。
<code>void write_seqlock_irqsave(seqlock_t *sl, unsigned long flags)</code>	保存中断状态，禁止本地中断，并获取写顺序锁。
<code>void write_sequnlock_irqrestore(seqlock_t *sl, unsigned long flags)</code>	将中断状态恢复到以前的状态，并且激活本地中断，释放写顺序锁。
<code>void write_seqlock_bh(seqlock_t *sl)</code>	关闭下半部，并获取写读锁。
<code>void write_sequnlock_bh(seqlock_t *sl)</code>	打开下半部，并释放写读锁。
顺序锁读操作	
<code>unsigned read_seqbegin(const seqlock_t *sl)</code>	读单元访问共享资源的时候调用此函数，此函数会返回顺序锁的序号。
<code>unsigned read_seqretry(const seqlock_t *sl, unsigned start)</code>	读结束以后调用此函数检查在读的过程中有没有对资源进行写操作，如果有的话就要重读

表 11.3.3.2 顺序锁 API 函数表

11.3.4 自旋锁使用注意事项

综合前面关于自旋锁的信息，我们需要在使用自旋锁的时候要注意以下几点：

①、因为在等待自旋锁的时候处于“自旋”状态，因此锁的持有时间不能太长，一定要短，否则的话会降低系统性能。如果临界区比较大，运行时间比较长的话要选择其他的并发处理方式，比如稍后要讲的信号量和互斥体。

②、自旋锁保护的临界区内不能调用任何可能导致线程休眠的 API 函数，否则的话可能导致死锁。

③、不能递归申请自旋锁，因为一旦通过递归的方式申请一个你正在持有的锁，那么你就必须“自旋”，等待锁被释放，然而你正处于“自旋”状态，根本没法释放锁。结果就是自己把自己锁死了！

④、在编写驱动程序的时候我们必须考虑到驱动的可移植性，因此不管你用的是单核的还是多核的 SOC，都将其当做多核 SOC 来编写驱动程序。

11.4 信号量

11.4.1 信号量简介

大家如果有学习过 FreeRTOS 或者 UCOS 的话就应该对信号量很熟悉，因为信号量是同步的一种方式。Linux 内核也提供了信号量机制，信号量常常用于控制对共享资源的访问。举一个很常见的例子，某个停车场有 100 个停车位，这 100 个停车位大家都可以用，对于大家来说这 100 个停车位就是共享资源。假设现在这个停车场正常运行，你要把车停到这个这个停车场肯定要先看一下现在停了多少车了？还有没有停车位？当前停车数量就是一个信号量，具体的停车数量就是这个信号量值，当这个值到 100 的时候说明停车场满了。停车场满的时你可以等一会看看有没有其他的车开出停车场，当有车开出停车场的时候停车数量就会减一，也就是说信号量减一，此时你就可以把车停进去了，你把车停进去以后停车数量就会加一，也就是信号量加一。这就是一个典型的使用信号量进行共享资源管理的案例，在这个案例中使用的就是计数型信号量。

相比于自旋锁，信号量可以使线程进入休眠状态，比如 A 与 B、C 合租了一套房子，这个房子只有一个厕所，一次只能一个人使用。某一天早上 A 去上厕所了，过了一会 B 也想上厕所，因为 A 在厕所里面，所以 B 只能等到 A 用完了才能进去。B 要么就一直在厕所门口等着，等 A 出来，这个时候就相当于自旋锁。B 也可以告诉 A，让 A 出来以后通知他一下，然后 B 继续回房间睡觉，这个时候相当于信号量。可以看出，使用信号量会提高处理器的使用效率，毕竟不用一直傻乎乎的在那里“自旋”等待。但是，信号量的开销要比自旋锁大，因为信号量使线程进入休眠状态以后会切换线程，切换线程就会有开销。总结一下信号量的特点：

①、因为信号量可以使等待资源线程进入休眠状态，因此适用于那些占用资源比较久的场合。

②、因此信号量不能用于中断中，因为信号量会引起休眠，中断不能休眠。

③、如果共享资源的持有时间比较短，那就不适合使用信号量了，因为频繁的休眠、切换线程引起的开销要远大于信号量带来的那点优势。

信号量有一个信号量值，相当于一个房子有 10 把钥匙，这 10 把钥匙就相当于信号量值为 10。因此，可以通过信号量来控制访问共享资源的访问数量，如果要想进房间，那就要先获取一把钥匙，信号量值减 1，直到 10 把钥匙都被拿走，信号量值为 0，这个时候就不允许任何人进入房间了，因为没钥匙了。如果有人从房间出来，那他要归还他所持有的那把钥匙，信号量值加 1，此时有 1 把钥匙了，那么可以允许进去一个人。相当于通过信号量控制访问资源的线程数，在初始化的时候将信号量值设置的大于 1，那么这个信号量就是计数型信号量，计数型信号量不能用于互斥访问，因为它允许多个线程同时访问共享资源。如果要互斥的访问共享资源

源那么信号量的值就不能大于 1，此时的信号量就是一个二值信号量。

11.4.2 信号量 API 函数

Linux 内核使用 semaphore 结构体表示信号量，结构体内容如下所示：

示例代码 11.4.2.1 semaphore 结构体

```
16 struct semaphore {
17     raw_spinlock_t    lock;
18     unsigned int      count;
19     struct list_head  wait_list;
20 };
```

要想使用信号量就得先定义，然后初始化信号量。有关信号量的 API 函数如表 11.4.2.1 所示：

函数	描述
DEFINE_SEMAPHORE(name)	定义一个信号量，并且设置信号量的值为 1。
void sema_init(struct semaphore *sem, int val)	初始化信号量 sem，设置信号量值为 val。
void down(struct semaphore *sem)	获取信号量，因为会导致休眠，因此不能在中断中使用。
int down_trylock(struct semaphore *sem);	尝试获取信号量，如果能获取到信号量就获取，并且返回 0。如果不能就返回非 0，并且不会进入休眠。
int down_interruptible(struct semaphore *sem)	获取信号量，和 down 类似，只是使用 down 进入休眠状态的线程不能被信号打断。而使用此函数进入休眠以后是可以被信号打断的。
void up(struct semaphore *sem)	释放信号量

表 11.4.2.1 信号量 API 函数

信号量的使用如下所示：

示例代码 11.4.2.2 信号量使用示例

```
struct semaphore sem; /* 定义信号量 */

sema_init(&sem, 1); /* 初始化信号量 */

down(&sem); /* 申请信号量 */
/* 临界区 */
up(&sem); /* 释放信号量 */
```

11.5 互斥体

11.5.1 互斥体简介

在 FreeRTOS 和 UCOS 中也有互斥体，将信号量的值设置为 1 就可以使用信号量进行互斥访问了，虽然可以通过信号量实现互斥，但是 Linux 提供了一个比信号量更专业的机制来进行互斥，它就是互斥体—mutex。互斥访问表示一次只有一个线程可以访问共享资源，不能递归申请互斥体。在我们编写 Linux 驱动的时候遇到需要互斥访问的地方建议使用 mutex。Linux 内核使用 mutex 结构体表示互斥体，定义如下(省略条件编译部分)：

示例代码 11.5.1.1 mutex 结构体

```
struct mutex {
    atomic_long_t    owner;
    spinlock_t      wait_lock;
};
```

在使用 mutex 之前要先定义一个 mutex 变量。在使用 mutex 的时候要注意如下几点:

- ①、mutex 可以导致休眠，因此不能在中断中使用 mutex，中断中只能使用自旋锁。
- ②、和信号量一样，mutex 保护的临界区可以调用引起阻塞的 API 函数。
- ③、因为一次只有一个线程可以持有 mutex，因此，必须由 mutex 的持有者释放 mutex。并且 mutex 不能递归上锁和解锁。

11.5.2 互斥体 API 函数

有关互斥体的 API 函数如表 11.5.2.1 所示:

函数	描述
DEFINE_MUTEX(name)	定义并初始化一个 mutex 变量。
void mutex_init(mutex *lock)	初始化 mutex。
void mutex_lock(struct mutex *lock)	获取 mutex，也就是给 mutex 上锁。如果获取不到就进休眠。
void mutex_unlock(struct mutex *lock)	释放 mutex，也就给 mutex 解锁。
int mutex_trylock(struct mutex *lock)	尝试获取 mutex，如果成功就返回 1，如果失败就返回 0。
int mutex_is_locked(struct mutex *lock)	判断 mutex 是否被获取，如果是的话就返回 1，否则返回 0。
int mutex_lock_interruptible(struct mutex *lock)	使用此函数获取信号量失败进入休眠以后可以被信号打断。

表 11.5.2.1 互斥体 API 函数

互斥体的使用如下所示:

示例代码 11.5.2.1 互斥体使用示例

```
1 struct mutex lock; /* 定义一个互斥体 */
2 mutex_init(&lock); /* 初始化互斥体 */
3
4 mutex_lock(&lock); /* 上锁 */
5 /* 临界区 */
6 mutex_unlock(&lock); /* 解锁 */
```

关于 Linux 中的并发和竞争就讲解到这里，Linux 内核还有很多其他的处理并发和竞争的机制，本章我们主要讲解了常用的原子操作、自旋锁、信号量和互斥体。以后我们在编写 Linux 驱动的时候就会频繁的使用到这几种机制，希望大家能够深入理解这几个常用的机制。

第十二章 Linux 并发与竞争实验

在上一章中我们学习了 Linux 下的并发与竞争，并且学习了四种常用的处理并发和竞争的机制：原子操作、自旋锁、信号量和互斥体。本章我们就通过四个实验来学习如何在驱动中使用这四种机制。

12.1 原子操作实验

本实验对应的例程路径为：[开发板光盘](#) → [01、程序源码](#) → [06、Linux 驱动例程源码](#) → [06_atomic](#)。

本例程我们在第十章的 `gpioled.c` 文件基础上完成。在本节我们使用原子操作来实现对 LED 这个设备的互斥访问，也就是一次只允许一个应用程序可以使用 LED 灯。

12.1.1 实验程序编写

1、修改设备树文件

因为本章实验是在第十章实验的基础上完成的，因此不需要对设备树做任何的修改。

2、LED 驱动修改

本节实验在第十章实验驱动文件 `gpioled.c` 的基础上修改而来。新建名为“06_atomic”的文件夹，然后在 06_atomic 文件夹里面创建 `vscode` 工程，工作区命名为“atomic”。将 5_gpioled 实验中的 `gpioled.c` 复制到 06_atomic 文件夹中，并且重命名为 `atomic.c`。本节实验重点就是使用 `atomic` 来实现一次只能允许一个应用访问 LED，所以我们只需要在 `atomic.c` 文件源码的基础上加上添加 `atomic` 相关代码即可，完成以后的 `atomic.c` 文件内容如下所示：

示例代码 12.1.1.1 atomic.c 文件代码段

```

1  #include <linux/types.h>
2  #include <linux/kernel.h>
3  #include <linux/delay.h>
4  #include <linux/ide.h>
5  #include <linux/init.h>
6  #include <linux/module.h>
7  #include <linux/errno.h>
8  #include <linux/gpio.h>
9  #include <linux/cdev.h>
10 #include <linux/device.h>
11 #include <linux/of.h>
12 #include <linux/of_address.h>
13 #include <linux/of_gpio.h>
14 #include <asm/mach/map.h>
15 #include <asm/uaccess.h>
16 #include <asm/io.h>
17 /*****
18 Copyright © ALIENTEK Co., Ltd. 1998-2029. All rights reserved.
19 文件名      : atomic.c
20 作者        : 正点原子 Linux 团队
21 版本        : V1.0
22 描述        : 原子操作实验，使用原子变量来实现对实现设备的互斥访问。
23 其他        : 无
24 论坛        : www.openedv.com
25 日志        : 初版 V1.0 2022/12/08 正点原子 Linux 团队创建
    
```

```

26  *****/
27  #define GPIOLED_CNT      1          /* 设备号个数 */
28  #define GPIOLED_NAME    "gpioled" /* 名字 */
29  #define LEDOFF          0          /* 关灯 */
30  #define LEDON           1          /* 开灯 */
31
32  /* gpioled 设备结构体 */
33  struct gpioled_dev{
34      dev_t devid;          /* 设备号 */
35      struct cdev cdev;    /* cdev */
36      struct class *class; /* 类 */
37      struct device *device; /* 设备 */
38      int major;          /* 主设备号 */
39      int minor;         /* 次设备号 */
40      struct device_node *nd; /* 设备节点 */
41      int led_gpio;      /* led 所使用的 GPIO 编号 */
42      atomic_t lock;     /* 原子变量 */
43  };
44
45  static struct gpioled_dev gpioled; /* led 设备 */
46
47
48  /*
49   * @description : 打开设备
50   * @param - inode : 传递给驱动的 inode
51   * @param - filp : 设备文件, file 结构体有个叫做 private_data 的成员变量
52   *                一般在 open 的时候将 private_data 指向设备结构体。
53   * @return      : 0 成功;其他 失败
54   */
55  static int led_open(struct inode *inode, struct file *filp)
56  {
57      /* 通过判断原子变量的值来检查 LED 有没有被别的应用使用 */
58      if (!atomic_dec_and_test(&gpioled.lock)) {
59          atomic_inc(&gpioled.lock); /* 小于 0 的话就加 1,使其原子变量等于 0 */
60          return -EBUSY;             /* LED 被使用, 返回忙 */
61      }
62
63      filp->private_data = &gpioled; /* 设置私有数据 */
64      return 0;
65  }
66
67  /*
68   * @description : 从设备读取数据

```

```

69  * @param - filp   : 要打开的设备文件 (文件描述符)
70  * @param - buf    : 返回给用户空间的数据缓冲区
71  * @param - cnt    : 要读取的数据长度
72  * @param - offt   : 相对于文件首地址的偏移
73  * @return         : 读取的字节数, 如果为负值, 表示读取失败
74  */
75  static ssize_t led_read(struct file *filp, char __user *buf, size_t
cnt, loff_t *offt)
76  {
77      return 0;
78  }
79
80  /*
81  * @description   : 向设备写数据
82  * @param - filp  : 设备文件, 表示打开的文件描述符
83  * @param - buf   : 要写给设备写入的数据
84  * @param - cnt   : 要写入的数据长度
85  * @param - offt  : 相对于文件首地址的偏移
86  * @return        : 写入的字节数, 如果为负值, 表示写入失败
87  */
88  static ssize_t led_write(struct file *filp, const char __user *buf,
size_t cnt, loff_t *offt)
89  {
90      int retvalue;
91      unsigned char databuf[1];
92      unsigned char ledstat;
93      struct gpioled_dev *dev = filp->private_data;
94
95      retvalue = copy_from_user(databuf, buf, cnt);
96      if(retvalue < 0) {
97          printk("kernel write failed!\r\n");
98          return -EFAULT;
99      }
100
101      ledstat = databuf[0];          /* 获取状态值 */
102
103      if(ledstat == LEDON) {
104          gpio_set_value(dev->led_gpio, 1); /* 打开 LED 灯 */
105      } else if(ledstat == LEDOFF) {
106          gpio_set_value(dev->led_gpio, 0); /* 关闭 LED 灯 */
107      }
108      return 0;
109  }

```

```

110
111 /*
112 * @description   : 关闭/释放设备
113 * @param - filp : 要关闭的设备文件(文件描述符)
114 * @return        : 0 成功;其他 失败
115 */
116 static int led_release(struct inode *inode, struct file *filp)
117 {
118     struct gpioled_dev *dev = filp->private_data;
119
120     /* 关闭驱动文件的时候释放原子变量 */
121     atomic_inc(&dev->lock);
122
123     return 0;
124 }
125
126 /* 设备操作函数 */
127 static struct file_operations gpioled_fops = {
128     .owner = THIS_MODULE,
129     .open = led_open,
130     .read = led_read,
131     .write = led_write,
132     .release = led_release,
133 };
134
135 /*
136 * @description   : 驱动出口函数
137 * @param         : 无
138 * @return        : 无
139 */
140 static int __init led_init(void)
141 {
142     int ret = 0;
143     const char *str;
144
145     /* 1、初始化原子变量 */
146     gpioled.lock = (atomic_t)ATOMIC_INIT(0);
147
148     /* 2、原子变量初始值为1 */
149     atomic_set(&gpioled.lock, 1);
150
151     /* 设置 LED 所使用的 GPIO */
152     /* 1、获取设备节点: gpioled */

```

```

153     gpioled.nd = of_find_node_by_path("/gpioled");
154     if(gpioled.nd == NULL) {
155         printk("gpioled node not find!\r\n");
156         return -EINVAL;
157     }
158
159     /* 2.读取 status 属性 */
160     ret = of_property_read_string(gpioled.nd, "status", &str);
161     if(ret < 0)
162         return -EINVAL;
163
164     if (strcmp(str, "okay"))
165         return -EINVAL;
166
167     /* 3、获取 compatible 属性值并进行匹配 */
168     ret = of_property_read_string(gpioled.nd, "compatible", &str);
169     if(ret < 0) {
170         printk("gpioled: Failed to get compatible property\n");
171         return -EINVAL;
172     }
173
174     if (strcmp(str, "alientek,led")) {
175         printk("gpioled: Compatible match failed\n");
176         return -EINVAL;
177     }
178
179     /* 4、获取设备树中的 gpio 属性,得到 LED 所使用的 LED 编号 */
180     gpioled.led_gpio = of_get_named_gpio(gpioled.nd, "led-gpio", 0);
181     if(gpioled.led_gpio < 0) {
182         printk("can't get led-gpio");
183         return -EINVAL;
184     }
185     printk("led-gpio num = %d\r\n", gpioled.led_gpio);
186
187     /* 5.向 gpio 子系统申请使用 GPIO */
188     ret = gpio_request(gpioled.led_gpio, "LED-GPIO");
189     if (ret) {
190         printk(KERN_ERR "gpioled: Failed to request led-gpio\n");
191         return ret;
192     }
193
194     /* 6、设置 PIO 为输出,并且输出高电平,默认关闭 LED 灯 */
195     ret = gpio_direction_output(gpioled.led_gpio, 1);
    
```

```

196     if(ret < 0) {
197         printk("can't set gpio!\r\n");
198     }
199
200     /* 注册字符设备驱动 */
201     /* 1、创建设备号 */
202     if (gpioled.major) {          /* 定义了设备号 */
203         gpioled.devid = MKDEV(gpioled.major, 0);
204         ret = register_chrdev_region(gpioled.devid, GPIOLED_CNT,
                                       GPIOLED_NAME);
205
206         if(ret < 0) {
207             pr_err("cannot register %s char driver [ret=%d]\n",
208                   GPIOLED_NAME, GPIOLED_CNT);
209             goto free_gpio;
210         }
211     } else {                      /* 没有定义设备号 */
212         ret = alloc_chrdev_region(&gpioled.devid, 0, GPIOLED_CNT,
213                                   GPIOLED_NAME);    /* 申请设备号 */
214
215         if(ret < 0) {
216             pr_err("%s Couldn't alloc_chrdev_region, ret=%d\r\n",
217                   GPIOLED_NAME, ret);
218             goto free_gpio;
219         }
220
221         gpioled.major = MAJOR(gpioled.devid); /* 获取分配号的主设备号 */
222         gpioled.minor = MINOR(gpioled.devid); /* 获取分配号的次设备号 */
223     }
224     printk("gpioled major=%d,minor=%d\r\n",gpioled.major,
225           gpioled.minor);
226
227     /* 2、初始化 cdev */
228     gpioled.cdev.owner = THIS_MODULE;
229     cdev_init(&gpioled.cdev, &gpioled_fops);
230
231     /* 3、添加一个 cdev */
232     cdev_add(&gpioled.cdev, gpioled.devid, GPIOLED_CNT);
233     if(ret < 0)
234         goto del_unregister;
235
236     /* 4、创建类 */
237     gpioled.class = class_create(THIS_MODULE, GPIOLED_NAME);
238     if (IS_ERR(gpioled.class)) {
239         goto del_cdev;
240     }

```

```

234
235     /* 5、创建设备 */
236     gpioled.device = device_create(gpioled.class, NULL,
                                   gpioled.devid, NULL, GPIOLED_NAME);
237     if (IS_ERR(gpioled.device)) {
238         goto destroy_class;
239     }
240     return 0;
241
242 destroy_class:
243     device_destroy(gpioled.class, gpioled.devid);
244 del_cdev:
245     cdev_del(&gpioled.cdev);
246 del_unregister:
247     unregister_chrdev_region(gpioled.devid, GPIOLED_CNT);
248 free_gpio:
249     gpio_free(gpioled.led_gpio);
250     return -EIO;
251 }
252
253 /*
254  * @description   : 驱动出口函数
255  * @param         : 无
256  * @return        : 无
257  */
258 static void __exit led_exit(void)
259 {
260     /* 注销字符设备驱动 */
261     cdev_del(&gpioled.cdev); /* 删除 cdev */
262     unregister_chrdev_region(gpioled.devid, GPIOLED_CNT);
263     device_destroy(gpioled.class, gpioled.devid); /* 注销设备 */
264     class_destroy(gpioled.class); /* 注销类 */
265     gpio_free(gpioled.led_gpio); /* 释放 GPIO */
266 }
267
268 module_init(led_init);
269 module_exit(led_exit);
270 MODULE_LICENSE("GPL");
271 MODULE_AUTHOR("ALIEN TEK");
272 MODULE_INFO(intree, "Y");
    
```

第 42 行, 原子变量 lock, 用来实现一次只能允许一个应用访问 LED 灯, led_init 驱动入口函数会将 lock 的值设置为 1。

第 57~61 行, 每次调用 `open` 函数打开驱动设备的时候先申请 `lock`, 如果申请成功的话就表示 LED 灯还没有被其他的应用使用, 如果申请失败就表示 LED 灯正在被其他的应用程序使用。每次打开驱动设备的时候先使用 `atomic_dec_and_test` 函数将 `lock` 减 1, 如果 `atomic_dec_and_test` 函数返回值为真就表示 `lock` 当前值为 0, 说明设备可以使用。如果 `atomic_dec_and_test` 函数返回值为假, 就表示 `lock` 当前值为负数(`lock` 值默认是 1), `lock` 值为负数的可能性只有一个, 那就是其他设备正在使用 LED。其他设备正在使用 LED 灯, 那么就只能退出了, 在退出之前调用函数 `atomic_inc` 将 `lock` 加 1, 因为此时 `lock` 的值被减成了负数, 必须要对其加 1, 将 `lock` 的值变为 0。

第 121 行, LED 灯使用完毕, 应用程序调用 `close` 函数关闭的驱动文件, `led_release` 函数执行, 调用 `atomic_inc` 释放 `lock`, 也就是将 `lock` 加 1。

第 146 行, 初始化原子变量 `lock`, 初始值设置为 0。

第 149 行, 原子变量 `lock` 设置为 1, 这样每次就只允许一个应用使用 LED 灯。

3、编写测试 APP

示例代码 12.1.1.2 atomicApp.c 文件代码

```

1  #include "stdio.h"
2  #include "unistd.h"
3  #include "sys/types.h"
4  #include "sys/stat.h"
5  #include "fcntl.h"
6  #include "stdlib.h"
7  #include "string.h"
8  /*****
9  Copyright © ALIENTEK Co., Ltd. 1998-2029. All rights reserved.
10  文件名      : atomicApp.c
11  作者       : 正点原子 Linux 团队
12  版本      : V1.0
13  描述      : 原子变量测试 APP, 测试原子变量能不能实现一次
14            只允许一个应用程序使用 LED。
15  其他      : 无
16  使用方法  : ./atomicApp /dev/gpioled 0 关闭 LED 灯
17            ./atomicApp /dev/gpioled 1 打开 LED 灯
18  论坛      : www.openedv.com
19  日志      : 初版 V1.0 2022/12/08 正点原子 Linux 团队创建
20  *****/
21
22  #define LEDOFF  0
23  #define LEDON   1
24
25  /*
26  * @description  : main 主程序
27  * @param - argc : argv 数组元素个数
28  * @param - argv : 具体参数
29  * @return      : 0 成功;其他 失败
    
```

```

30  */
31  int main(int argc, char *argv[])
32  {
33      int fd, retvalue;
34      char *filename;
35      unsigned char cnt = 0;
36      unsigned char databuf[1];
37
38      if(argc != 3){
39          printf("Error Usage!\r\n");
40          return -1;
41      }
42
43      filename = argv[1];
44
45      /* 打开 led 驱动 */
46      fd = open(filename, O_RDWR);
47      if(fd < 0){
48          printf("file %s open failed!\r\n", argv[1]);
49          return -1;
50      }
51
52      databuf[0] = atoi(argv[2]); /* 要执行的操作: 打开或关闭 */
53
54      /* 向/dev/gpioled 文件写入数据 */
55      retvalue = write(fd, databuf, sizeof(databuf));
56      if(retvalue < 0){
57          printf("LED Control Failed!\r\n");
58          close(fd);
59          return -1;
60      }
61
62      /* 模拟占用 25S LED */
63      while(1) {
64          sleep(5);
65          cnt++;
66          printf("App running times:%d\r\n", cnt);
67          if(cnt >= 5) break;
68      }
69
70      printf("App running finished!");
71      retvalue = close(fd); /* 关闭文件 */
72      if(retvalue < 0){
    
```

```

73     printf("file %s close failed!\r\n", argv[1]);
74     return -1;
75 }
76 return 0;
77 }
```

12.1.2 运行测试

1、编译驱动程序

编写 Makefile 文件，本章实验的 Makefile 文件和第五章实验基本一样，只是将 obj-m 变量的值改为 atomic.o，Makefile 内容如下所示：

示例代码 12.1.2.1 Makefile 文件

```

1 KERNELDIR := /home/alientek/rk3568_linux_sdk/kernel
.....
4 obj-m := atomic.o
.....
11 clean:
12 $(MAKE) -C $(KERNELDIR) M=$(CURRENT_PATH) clean
```

第 4 行，设置 obj-m 变量的值为 atomic.o。

输入如下命令编译出驱动模块文件：

```
make ARCH=arm64 //ARCH=arm64 必须指定，否则编译会失败
```

编译成功以后就会生成一个名为“atomic.ko”的驱动模块文件。

2、编译测试 APP

输入如下命令编译测试 atomicApp.c 这个测试程序：

```
/opt/atk-dlrk356x-toolchain/bin/aarch64-buildroot-linux-gnu-gcc atomicApp.c -o atomicApp
```

编译成功以后就会生成 ledApp 这个应用程序。

3、运行测试

在 Ubuntu 中将上一小节编译出来的 atomic.ko 和 atomicApp 这两个文件通过 adb 命令发送到开发板的/lib/modules/4.19.232 目录下，命令如下：

```
adb push atomic.ko atomicApp /lib/modules/4.19.232
```

发送成功以后进入到开发板目录 lib/modules/4.19.232 中，输入如下命令加载 atomic.ko 驱动模块：

```
depmod //第一次加载驱动的时候需要运行此命令
```

```
modprobe atomic //加载驱动
```

驱动加载成功以后就可以使用 atomicApp 软件来测试驱动是否工作正常，输入如下命令以后台运行模式打开 LED 灯，“&”表示在后台运行 atomicApp 这个软件：

```
./atomicApp /dev/gpioled 1 & //打开 LED 灯
```

输入上述命令以后观察开发板上的绿色 LED 灯是否点亮，然后每隔 5 秒都会输出一行“App running times”，如图 12.1.2.1 所示：

```

root@ATK-DLRK356X:/lib/modules/4.19.232# ./atomicApp /dev/gpioled 1 &
[1] 1076
root@ATK-DLRK356X:/lib/modules/4.19.232# App running times:1
App running times:2
App running times:3
```

图 12.1.2.1 打开 LED 灯

从图 12.1.2.1 可以看出, atomicApp 运行正常, 输出了“App running times:1”和“App running times:2”等字符串, 这就是模拟 25S 占用, 说明 atomicApp 这个软件正在使用 LED 灯。此时再输入如下命令关闭 LED 灯:

```
./atomicApp /dev/gpioled 0 //关闭 LED 灯
```

输入上述命令以后会发现如图 12.1.2.2 所示输入信息:

```
root@ATK-DLRK356X:/lib/modules/4.19.232# ./atomicApp /dev/gpioled 0
file /dev/gpioled open failed!
root@ATK-DLRK356X:/lib/modules/4.19.232#
```

图 12.1.2.2 关闭 LED 灯

从图 12.1.2.2 可以看出, 打开/dev/gpioled 失败! 原因是在图 12.1.2.1 中运行的 atomicAPP 软件正在占用/dev/gpioled, 如果再次运行 atomicApp 软件去操作/dev/gpioled 肯定会失败。必须等待图 12.1.2.1 中的 atomicApp 运行结束, 也就是 25S 结束以后其他软件才能去操作/dev/gpioled。这个就是采用原子变量实现一次只能有一个应用程序访问 LED 灯。

如果要卸载驱动的话输入如下命令即可:

```
rmmmod atomic.ko
```

12.2 自旋锁实验

本实验对应的例程路径为: [开发板光盘](#)→01、程序源码→Linux 驱动例程源码→07_spinlock。

上一节我们使用原子变量实现了一次只能有一个应用程序访问 LED 灯, 本节我们使用自旋锁来实现此功能。在使用自旋锁之前, 先回顾一下自旋锁的使用注意事项:

①、自旋锁保护的临界区要尽可能的短, 因此在 open 函数中申请自旋锁, 然后在 release 函数中释放自旋锁的方法就不可取。我们可以使用一个变量来表示设备的使用情况, 如果设备被使用了那么变量就加一, 设备被释放以后变量就减 1, 我们只需要使用自旋锁保护这个变量即可。

②、考虑驱动的兼容性, 合理的选择 API 函数。

综上所述, 在本节例程中, 我们通过定义一个变量 dev_stats 表示设备的使用情况, dev_stats 为 0 的时候表示设备没有被使用, dev_stats 大于 0 的时候表示设备被使用。驱动 open 函数中先判断 dev_stats 是否为 0, 也就是判断设备是否可用, 如果为 0 的话就使用设备, 并且将 dev_stats 加 1, 表示设备被使用了。使用完以后在 release 函数中将 dev_stats 减 1, 表示设备没有被使用了。因此真正实现设备互斥访问的是变量 dev_stats, 但是我们要使用自旋锁对 dev_stats 来做保护。

12.2.1 实验程序编写

1、修改设备树文件

本章实验是在上一节实验的基础上完成的, 同样不需要对设备树做任何的修改。

2、LED 驱动修改

本节实验在第上一节实验驱动文件 atomic.c 的基础上修改而来。新建名为“07_spinlock”的文件夹, 然后在 07_spinlock 文件夹里面创建 vscode 工程, 工作区命名为“spinlock”。将 7_atomic 实验中的 atomic.c 复制到 07_spinlock 文件夹中, 并且重命名为 spinlock.c。将原来使用 atomic 的地方换为 spinlock 即可, 其他代码不需要修改, 完成以后的 spinlock.c 文件内容如下所示(有省略):

示例代码 12.2.1.1 spinlock.c 文件代码

```

1  #include <linux/types.h>
2  #include <linux/kernel.h>
3  #include <linux/delay.h>
...
17 /*****
18 Copyright © ALIENTEK Co., Ltd. 1998-2029. All rights reserved.
19 文件名      : spinlock.c
20 作者        : 正点原子 Linux 团队
21 版本        : V1.0
22 描述        : 自旋锁实验, 使用自旋锁来实现对实现设备的互斥访问。
23 其他        : 无
24 论坛        : www.openedv.com
25 日志        : 初版 V1.0 2022/12/08 正点原子 Linux 团队创建
26 *****/
27 #define GPIOLED_CNT      1          /* 设备号个数 */
28 #define GPIOLED_NAME     "gpioled" /* 名字 */
29 #define LEDOFF           0          /* 关灯 */
30 #define LEDON            1          /* 开灯 */
31
32 /* gpioled 设备结构体 */
33 struct gpioled_dev{
34     dev_t devid;          /* 设备号 */
35     struct cdev cdev;     /* cdev */
36     struct class *class;  /* 类 */
37     struct device *device; /* 设备 */
38     int major;           /* 主设备号 */
39     int minor;           /* 次设备号 */
40     struct device_node *nd; /* 设备节点 */
41     int led_gpio;        /* led 所使用的 GPIO 编号 */
42     int dev_stats;       /* 设备使用状态, 0, 设备未使用;>0, 设备已经被使用 */
43     spinlock_t lock;     /* 自旋锁 */
44 };
45
46 static struct gpioled_dev gpioled; /* led 设备 */
47
48
49 /*
50 * @description   : 打开设备
51 * @param - inode : 传递给驱动的 inode
52 * @param - filp  : 设备文件, file 结构体有个叫做 private_data 的成员变量
53 *                  一般在 open 的时候将 private_data 指向设备结构体。
54 * @return        : 0 成功;其他 失败
    
```

```

55  */
56 static int led_open(struct inode *inode, struct file *filp)
57 {
58     unsigned long flags;
59     filp->private_data = &gpioled; /* 设置私有数据 */
60
61     spin_lock_irqsave(&gpioled.lock, flags); /* 上锁 */
62     if (gpioled.dev_stats) {                /* 如果设备被使用了 */
63         spin_unlock_irqrestore(&gpioled.lock, flags); /* 解锁 */
64         return -EBUSY;
65     }
66     gpioled.dev_stats++; /* 如果设备没有打开, 那么就标记已经打开了 */
67     spin_unlock_irqrestore(&gpioled.lock, flags); /* 解锁 */
68
69     return 0;
70 }
71
72 /*
73  * @description   : 从设备读取数据
74  * @param - filp  : 要打开的设备文件(文件描述符)
75  * @param - buf   : 返回给用户空间的数据缓冲区
76  * @param - cnt   : 要读取的数据长度
77  * @param - offt  : 相对于文件首地址的偏移
78  * @return        : 读取的字节数, 如果为负值, 表示读取失败
79  */
80 static ssize_t led_read(struct file *filp, char __user *buf,
81                        size_t cnt, loff_t *offt)
82 {
83     return 0;
84 }
85 /*
86  * @description   : 向设备写数据
87  * @param - filp  : 设备文件, 表示打开的文件描述符
88  * @param - buf   : 要写给设备写入的数据
89  * @param - cnt   : 要写入的数据长度
90  * @param - offt  : 相对于文件首地址的偏移
91  * @return        : 写入的字节数, 如果为负值, 表示写入失败
92  */
93 static ssize_t led_write(struct file *filp, const char __user *buf,
94                        size_t cnt, loff_t *offt)
95 {
96     int retvalue;

```

```

96     unsigned char databuf[1];
97     unsigned char ledstat;
98     struct gpioled_dev *dev = filp->private_data;
99
100    retvalue = copy_from_user(databuf, buf, cnt);
101    if(retvalue < 0) {
102        printk("kernel write failed!\r\n");
103        return -EFAULT;
104    }
105
106    ledstat = databuf[0];        /* 获取状态值 */
107
108    if(ledstat == LEDON) {
109        gpio_set_value(dev->led_gpio, 1);    /* 打开LED灯 */
110    } else if(ledstat == LEDOFF) {
111        gpio_set_value(dev->led_gpio, 0);    /* 关闭LED灯 */
112    }
113    return 0;
114 }
115
116 /*
117  * @description   : 关闭/释放设备
118  * @param - filp  : 要关闭的设备文件(文件描述符)
119  * @return        : 0 成功;其他 失败
120  */
121 static int led_release(struct inode *inode, struct file *filp)
122 {
123     unsigned long flags;
124     struct gpioled_dev *dev = filp->private_data;
125
126     /* 关闭驱动文件的时候将 dev_stats 减1 */
127     spin_lock_irqsave(&dev->lock, flags);    /* 上锁 */
128     if (dev->dev_stats) {
129         dev->dev_stats--;
130     }
131     spin_unlock_irqrestore(&dev->lock, flags);/* 解锁 */
132
133     return 0;
134 }
135
136 /* 设备操作函数 */
137 static struct file_operations gpioled_fops = {
138     .owner = THIS_MODULE,

```

```

139     .open = led_open,
140     .read = led_read,
141     .write = led_write,
142     .release = led_release,
143 };
144
145 /*
146 * @description   : 驱动出口函数
147 * @param         : 无
148 * @return        : 无
149 */
150 static int __init led_init(void)
151 {
152     int ret = 0;
153     const char *str;
154
155     /* 初始化自旋锁 */
156     spin_lock_init(&gpioled.lock);
157
158     /* 设置 LED 所使用的 GPIO */
159     /* 1、获取设备节点: gpioled */
160     gpioled.nd = of_find_node_by_path("/gpioled");
161     if(gpioled.nd == NULL) {
162         printk("gpioled node not find!\r\n");
163         return -EINVAL;
164     }
165
166     /* 2.读取 status 属性 */
167     ret = of_property_read_string(gpioled.nd, "status", &str);
168     if(ret < 0)
169         return -EINVAL;
170
171     if (strcmp(str, "okay"))
172         return -EINVAL;
173
174     /* 3、获取 compatible 属性值并进行匹配 */
175     ret = of_property_read_string(gpioled.nd, "compatible", &str);
176     if(ret < 0) {
177         printk("gpioled: Failed to get compatible property\n");
178         return -EINVAL;
179     }
180
181     if (strcmp(str, "alientek,led")) {

```

```

182     printk("gpioled: Compatible match failed\n");
183     return -EINVAL;
184 }
185
186 /* 4、获取设备树中的 gpio 属性, 得到 LED 所使用的 LED 编号 */
187 gpioled.led_gpio = of_get_named_gpio(gpioled.nd, "led-gpio", 0);
188 if(gpioled.led_gpio < 0) {
189     printk("can't get led-gpio");
190     return -EINVAL;
191 }
192 printk("led-gpio num = %d\r\n", gpioled.led_gpio);
193
194 /* 5.向 gpio 子系统申请使用 GPIO */
195 ret = gpio_request(gpioled.led_gpio, "LED-GPIO");
196 if (ret) {
197     printk(KERN_ERR "gpioled: Failed to request led-gpio\n");
198     return ret;
199 }
200
201 /* 6、设置 PI0 为输出, 并且输出高电平, 默认关闭 LED 灯 */
202 ret = gpio_direction_output(gpioled.led_gpio, 1);
203 if(ret < 0) {
204     printk("can't set gpio!\r\n");
205 }
206
207 /* 注册字符设备驱动 */
208 /* 1、创建设备号 */
209 if (gpioled.major) { /* 定义了设备号 */
210     gpioled.devid = MKDEV(gpioled.major, 0);
211     ret = register_chrdev_region(gpioled.devid, GPIOLED_CNT,
212                                 GPIOLED_NAME);
213
214     if(ret < 0) {
215         pr_err("cannot register %s char driver [ret=%d]\n",
216               GPIOLED_NAME, GPIOLED_CNT);
217         goto free_gpio;
218     }
219 } else { /* 没有定义设备号 */
220     ret = alloc_chrdev_region(&gpioled.devid, 0, GPIOLED_CNT,
221                               GPIOLED_NAME); /* 申请设备号 */
222
223     if(ret < 0) {
224         pr_err("%s Couldn't alloc_chrdev_region, ret=%d\r\n",
225               GPIOLED_NAME, ret);
226         goto free_gpio;
227     }
228 }

```

```

221     }
222     gpioled.major = MAJOR(gpioled.devid); /* 获取分配号的主设备号 */
223     gpioled.minor = MINOR(gpioled.devid); /* 获取分配号的次设备号 */
224     }
225     printk("gpioled major=%d,minor=%d\r\n",gpioled.major,
            gpioled.minor);
226
227     /* 2、初始化 cdev */
228     gpioled.cdev.owner = THIS_MODULE;
229     cdev_init(&gpioled.cdev, &gpioled_fops);
230
231     /* 3、添加一个 cdev */
232     cdev_add(&gpioled.cdev, gpioled.devid, GPIOLED_CNT);
233     if(ret < 0)
234         goto del_unregister;
235
236     /* 4、创建类 */
237     gpioled.class = class_create(THIS_MODULE, GPIOLED_NAME);
238     if (IS_ERR(gpioled.class)) {
239         goto del_cdev;
240     }
241
242     /* 5、创建设备 */
243     gpioled.device = device_create(gpioled.class, NULL,
                                   gpioled.devid, NULL, GPIOLED_NAME);
244     if (IS_ERR(gpioled.device)) {
245         goto destroy_class;
246     }
247     return 0;
248
249 destroy_class:
250     device_destroy(gpioled.class, gpioled.devid);
251 del_cdev:
252     cdev_del(&gpioled.cdev);
253 del_unregister:
254     unregister_chrdev_region(gpioled.devid, GPIOLED_CNT);
255 free_gpio:
256     gpio_free(gpioled.led_gpio);
257     return -EIO;
258 }
259
260 /*
261 * @description : 驱动出口函数
    
```

```

262 * @param      : 无
263 * @return     : 无
264 */
265 static void __exit led_exit(void)
266 {
267     /* 注销字符设备驱动 */
268     cdev_del(&gpioled.cdev); /* 删除 cdev */
269     unregister_chrdev_region(gpioled.devid, GPIOLED_CNT);
270     device_destroy(gpioled.class, gpioled.devid); /* 注销设备 */
271     class_destroy(gpioled.class); /* 注销类 */
272     gpio_free(gpioled.led_gpio); /* 释放 GPIO */
273 }
274
275 module_init(led_init);
276 module_exit(led_exit);
277 MODULE_LICENSE("GPL");
278 MODULE_AUTHOR("ALIEN TEK");
279 MODULE_INFO(intree, "Y");
    
```

第 42 行, `dev_stats` 表示设备状态, 如果为 0 的话表示设备还没有被使用, 如果大于 0 的话就表示设备已经被使用了。

第 43 行, 定义自旋锁变量 `lock`。

第 61~67 行, 使用自旋锁实现对设备的互斥访问, 第 61 行调用 `spin_lock_irqsave` 函数获取锁, 为了考虑到驱动兼容性, 这里并没有使用 `spin_lock` 函数来获取锁。第 62 行判断 `dev_stats` 是否大于 0, 如果是的话表示设备已经被使用了, 那么就调用 `spin_unlock_irqrestore` 函数释放锁, 并且返回 `-EBUSY`。如果设备没有被使用的话就在第 66 行将 `dev_stats` 加 1, 表示设备要被使用了, 然后调用 `spin_unlock_irqrestore` 函数释放锁。自旋锁的工作就是保护 `dev_stats` 变量, 真正实现对设备互斥访问的是 `dev_stats`。

第 127~131 行, 在 `release` 函数中将 `dev_stats` 减 1, 表示设备被释放了, 可以被其他的应用程序使用。将 `dev_stats` 减 1 的时候需要自旋锁对其进行保护。

第 156 行, 在驱动入口函数 `led_init` 中调用 `spin_lock_init` 函数初始化自旋锁。

3、编写测试 APP

测试 APP 使用 12.1.1 小节中的 `atomicApp.c` 即可, 将 `06_atomic` 中的 `atomicApp.c` 文件拷贝到本例程中, 并将 `atomicApp.c` 重命名为 `spinlockApp.c` 即可。

12.2.2 运行测试

1、编译驱动程序

编写 Makefile 文件，本章实验的 Makefile 文件和第五章实验基本一样，只是将 obj-m 变量的值改为 spinlock.o，Makefile 内容如下所示：

示例代码 12.2.2.1 Makefile 文件

```
1 KERNELDIR := /home/alientek/rk3568_linux_sdk/kernel
.....
4 obj-m := spinlock.o
.....
11 clean:
12 $(MAKE) -C $(KERNELDIR) M=$(CURRENT_PATH) clean
```

第 4 行，设置 obj-m 变量的值为 spinlock.o。

输入如下命令编译出驱动模块文件：

```
make ARCH=arm64 //ARCH=arm64 必须指定，否则编译会失败
编译成功以后就会生成一个名为“spinlock.ko”的驱动模块文件。
```

2、编译测试 APP

输入如下命令编译测试 spinlockApp.c 这个测试程序：

```
/opt/atk-dlrk356x-toolchain/bin/aarch64-buildroot-linux-gnu-gcc spinlockApp.c -o spinlockApp
编译成功以后就会生成 spinlockApp 这个应用程序。
```

3、运行测试

在 Ubuntu 中将上一小节编译出来的 spinlock.ko 和 spinlockApp 这两个文件通过 adb 命令发送到开发板的/lib/modules/4.19.232 目录下，命令如下：

```
adb push spinlock.ko spinlockApp /lib/modules/4.19.232
```

发送成功以后进入到开发板目录 lib/modules/4.19.232 中，输入如下命令加载 atomic.ko 驱动模块：

```
depmod //第一次加载驱动的时候需要运行此命令
modprobe spinlock //加载驱动
```

驱动加载成功以后就可以使用 spinlockApp 软件测试驱动是否工作正常，测试方法和 12.1.2 小节中一样，先输入如下命令让 spinlockAPP 软件模拟占用 25S 的 LED 灯：

```
./spinlockApp /dev/gpioled 1& //打开 LED 灯
```

紧接着再输入如下命令关闭 LED 灯：

```
./spinlockApp /dev/gpioled 0 //关闭 LED 灯
```

看一下能不能关闭 LED 灯，驱动正常工作的话并不会马上关闭 LED 灯，会提示你“file /dev/gpioled open failed!”，必须等待第一个 spinlockApp 软件运行完成(25S 计时结束)才可以再次操作 LED 灯。

如果要卸载驱动的话输入如下命令即可：

```
rmmmod spinlock.ko
```

12.3 信号量实验

本节我们使用信号量来实现一次只能有一个应用程序访问 LED 灯，信号量可以导致休眠，因此信号量保护的临界区没有运行时间限制，可以在驱动的 open 函数申请信号量，然后在

release 函数中释放信号量。但是信号量不能用在中断中，本节实验我们不会在中断中使用信号量。

12.3.1 实验程序编写

1、修改设备树文件

本章实验是在上一节实验的基础上完成的，同样不需要对设备树做任何的修改。

2、LED 驱动修改

本节实验在第上一节实验驱动文件 spinlock.c 的基础上修改而来。新建名为“08_semaphore”的文件夹，然后在 08_semaphore 文件夹里面创建 vscode 工程，工作区命名为“semaphore”。将 07_spinlock 实验中的 spinlock.c 复制到 08_semaphore 文件夹中，并且重命名为 semaphore.c。将原来使用到自旋锁的地方换为信号量即可，其他的内容基本不变，完成以后的 semaphore.c 文件内容如下所示(有省略):

示例代码 12.3.1.1 semaphore.c 文件代码

```

1  #include <linux/types.h>
.....
14 #include <linux/semaphore.h>
15 #include <asm/mach/map.h>
16 #include <asm/uaccess.h>
17 #include <asm/io.h>
18 /*****
19 Copyright © ALIENTEK Co., Ltd. 1998-2029. All rights reserved.
20 文件名      : semaphore.c
21 作者        : 正点原子 Linux 团队
22 版本        : V1.0
23 描述        : 信号量实验，使用信号量来实现对实现设备的互斥访问。
24 其他        : 无
25 论坛        : www.openedv.com
26 日志        : 初版 V1.0 2022/12/08 正点原子 Linux 团队创建
27 *****/
28 #define GPIOLED_CNT      1          /* 设备号个数 */
29 #define GPIOLED_NAME     "gpioled" /* 名字      */
30 #define LEDOFF           0          /* 关灯      */
31 #define LEDON            1          /* 开灯      */
32
33 /* gpioled 设备结构体 */
34 struct gpioled_dev{
35     dev_t devid;          /* 设备号    */
36     struct cdev cdev;     /* cdev      */
37     struct class *class;  /* 类        */
38     struct device *device; /* 设备      */
39     int major;           /* 主设备号  */
40     int minor;           /* 次设备号  */
    
```

```

41     struct device_node *nd; /* 设备节点 */
42     int led_gpio;          /* led 所使用的 GPIO 编号 */
43     struct semaphore sem; /* 信号量 */
44 };
45
46 static struct gpioled_dev gpioled; /* led 设备 */
47
48
49 /*
50  * @description      : 打开设备
51  * @param - inode    : 传递给驱动的 inode
52  * @param - filp     : 设备文件, file 结构体有个叫做 private_data 的成员变
53  *                   量一般在 open 的时候将 private_data 指向设备结构体。
54  * @return           : 0 成功;其他 失败
55  */
56 static int led_open(struct inode *inode, struct file *filp)
57 {
58     filp->private_data = &gpioled; /* 设置私有数据 */
59
60     /* 获取信号量 */
61     if (down_interruptible(&gpioled.sem)) {
62         return -ERESTARTSYS;
63     }
64 #if 0
65     down(&gpioled.sem); /* 不能被信号打断 */
66 #endif
67
68     return 0;
69 }
70
71 .....
120 static int led_release(struct inode *inode, struct file *filp)
121 {
122     struct gpioled_dev *dev = filp->private_data;
123
124     up(&dev->sem); /* 释放信号量, 信号量 count 值加 1 */
125     return 0;
126 }
127
128 /* 设备操作函数 */
129 static struct file_operations gpioled_fops = {
130     .owner = THIS_MODULE,
131     .open = led_open,
132     .read = led_read,

```

```

133     .write = led_write,
134     .release = led_release,
135 };
136
137 /*
138  * @description   : 驱动出口函数
139  * @param         : 无
140  * @return        : 无
141  */
142 static int __init led_init(void)
143 {
144     int ret = 0;
145     const char *str;
146
147     /* 初始化信号量 */
148     sema_init(&gpioled.sem, 1);
149     .....
150 }
151
152 /*
153  * @description   : 驱动出口函数
154  * @param         : 无
155  * @return        : 无
156  */
157 static void __exit led_exit(void)
158 {
159     /* 注销字符设备驱动 */
160     cdev_del(&gpioled.cdev); /* 删除 cdev */
161     unregister_chrdev_region(gpioled.devid, GPIOLED_CNT);
162     device_destroy(gpioled.class, gpioled.devid); /* 注销设备 */
163     class_destroy(gpioled.class); /* 注销类 */
164     gpio_free(gpioled.led_gpio); /* 释放 GPIO */
165 }
166
167 module_init(led_init);
168 module_exit(led_exit);
169 MODULE_LICENSE("GPL");
170 MODULE_AUTHOR("ALIENTEK");
171 MODULE_INFO(intree, "Y");
    
```

第 14 行, 要使用信号量必须添加<linux/semaphore.h>头文件。

第 43 行, 在设备结构体中添加一个信号量成员变量 sem。

第 60~66 行, 在 open 函数中申请信号量, 可以使用 down 函数, 也可以使用 down_interruptible 函数。如果信号量值大于等于 1 就表示可用, 那么应用程序就会开始使用 LED 灯。如果信号量

值为 0 就表示应用程序不能使用 LED 灯, 此时应用程序就会进入到休眠状态。等到信号量值大于 1 的时候应用程序就会唤醒, 申请信号量, 获取 LED 灯使用权。

第 124 行, 在 `release` 函数中调用 `up` 函数释放信号量, 这样其他因为没有得到信号量而进入休眠状态的应用程序就会唤醒, 获取信号量。

第 148 行, 在驱动入口函数中调用 `sema_init` 函数初始化信号量 `sem` 的值为 1, 相当于 `sem` 是个二值信号量。

总结一下, 当信号量 `sem` 为 1 的时候表示 LED 灯还没有被使用, 如果应用程序 A 要使用 LED 灯, 先调用 `open` 函数打开 `/dev/gpioled`, 这个时候会获取信号量 `sem`, 获取成功以后 `sem` 的值减 1 变为 0。如果此时应用程序 B 也要使用 LED 灯, 调用 `open` 函数打开 `/dev/gpioled` 就会因为信号量无效(值为 0)而进入休眠状态。当应用程序 A 运行完毕, 调用 `close` 函数关闭 `/dev/gpioled` 的时候就会释放信号量 `sem`, 此时信号量 `sem` 的值就会加 1, 变为 1。信号量 `sem` 再次有效, 表示其他应用程序可以使用 LED 灯了, 此时在休眠状态的应用程序 A 就会获取到信号量 `sem`, 获取成功以后就开始使用 LED 灯。

3、编写测试 APP

测试 APP 使用 12.1.1 小节中的 `atomicApp.c` 即可, 将 `06_atomic` 中的 `atomicApp.c` 文件到本例程中, 并将 `atomicApp.c` 重命名为 `semaApp.c` 即可。

12.3.2 运行测试

1、编译驱动程序

编写 Makefile 文件, 本章实验的 Makefile 文件和第五章实验基本一样, 只是将 `obj-m` 变量的值改为 `semaphore.o`, Makefile 内容如下所示:

示例代码 12.1.3.1 Makefile 文件

```
1 KERNELDIR := /home/alientek/rk3568_linux_sdk/kernel
.....
4 obj-m := semaphore.o
.....
11 clean:
12 $(MAKE) -C $(KERNELDIR) M=$(CURRENT_PATH) clean
```

第 4 行, 设置 `obj-m` 变量的值为 `semaphore.o`。

输入如下命令编译出驱动模块文件:

```
make ARCH=arm64 //ARCH=arm64 必须指定, 否则编译会失败
```

编译成功以后就会生成一个名为“`semaphore.ko`”的驱动模块文件。

2、编译测试 APP

输入如下命令编译测试 `semaApp.c` 这个测试程序:

```
/opt/atk-dlrk356x-toolchain/bin/aarch64-buildroot-linux-gnu-gcc semaApp.c -o semaApp
```

编译成功以后就会生成 `semaApp` 这个应用程序。

3、运行测试

在 Ubuntu 中将上一小节编译出来的 `semaphore.ko` 和 `semaApp` 这两个文件通过 `adb` 命令发送到开发板的 `/lib/modules/4.19.232` 目录下, 命令如下:

```
adb push semaphore.ko semaApp /lib/modules/4.19.232
```

发送成功以后进入到开发板目录 `lib/modules/4.19.232` 中, 输入如下命令加载 `atomic.ko` 驱

动模块:

```
depmod //第一次加载驱动的时候需要运行此命令
modprobe semaphore //加载驱动
```

驱动加载成功以后就可以使用 semaApp 软件测试驱动是否工作正常,测试方法和 12.1.2 小节中一样,先输入如下命令让 semaApp 软件模拟占用 25S 的 LED 灯:

```
./semaApp /dev/gpioled 1 & //打开 LED 灯
```

紧接着再输入如下命令关闭 LED 灯:

```
./semaApp /dev/gpioled 0 & //关闭 LED 灯
```

注意两个命令都是运行在后台,第一条命令先获取到信号量,因此可以操作 LED 灯,将 LED 灯打开,并且占有 25S。第二条命令因为获取信号量失败而进入休眠状态,等待第一条命令运行完毕并释放信号量以后才拥有 LED 灯使用权,将 LED 灯关闭,运行结果如图 12.3.2.1 所示:



图 12.3.2.1 命令运行过程

如果要卸载驱动的话输入如下命令即可:

```
rmmmod semaphore.ko
```

12.4 互斥体实验

前面我们使用原子操作、自旋锁和信号量实现了对 LED 灯的互斥访问,但是最适合互斥的就是互斥体 mutex 了。本节我们来学习一下如何使用 mutex 实现对 LED 灯的互斥访问。

12.4.1 实验程序编写

1、修改设备树文件

本章实验是在上一节实验的基础上完成的,同样不需要对设备树做任何的修改。

2、LED 驱动修改

本节实验在第上一节实验驱动文件 semaphore.c 的基础上修改而来。新建名为“10_mutex”的文件夹,然后在 10_mutex 文件夹里面创建 vscode 工程,工作区命名为“mutex”。将 9_semaphore 实验中的 semaphore.c 复制到 10_mutex 文件夹中,并且重命名为 mutex.c。将原来使用到信号量的地方换为 mutex 即可,其他的内容基本不变,完成以后的 mutex.c 文件内容如下所示(有省略):

示例代码 12.4.1.1 mutex.c 文件代码

```
1 #include <linux/types.h>
.....
18 /*****
```

```

19 Copyright © ALIENTEK Co., Ltd. 1998-2029. All rights reserved.
20 文件名      : mutex.c
21 作者        : 正点原子 Linux 团队
22 版本        : V1.0
23 描述        : 互斥体实验, 使用互斥体来实现对实现设备的互斥访问。
24 其他        : 无
25 论坛        : www.openedv.com
26 日志        : 初版 V1.0 2022/12/08 正点原子 Linux 团队创建
27 *****/
28 #define GPIOLED_CNT      1          /* 设备号个数 */
29 #define GPIOLED_NAME     "gpioled" /* 名字 */
30 #define LEDOFF           0          /* 关灯 */
31 #define LEDON            1          /* 开灯 */
32
33 /* gpioled 设备结构体 */
34 struct gpioled_dev{
35     dev_t devid;          /* 设备号 */
36     struct cdev cdev;     /* cdev */
37     struct class *class; /* 类 */
38     struct device *device; /* 设备 */
39     int major;           /* 主设备号 */
40     int minor;          /* 次设备号 */
41     struct device_node *nd; /* 设备节点 */
42     int led_gpio;       /* led 所使用的 GPIO 编号 */
43     struct mutex lock;  /* 互斥体 */
44 };
45
46 static struct gpioled_dev gpioled; /* led 设备 */
47
48
49 /*
50 * @description   : 打开设备
51 * @param - inode : 传递给驱动的 inode
52 * @param - filp  : 设备文件, file 结构体有个叫做 private_data 的成员变量
53 *                一般在 open 的时候将 private_data 指向设备结构体。
54 * @return        : 0 成功;其他 失败
55 */
56 static int led_open(struct inode *inode, struct file *filp)
57 {
58     filp->private_data = &gpioled; /* 设置私有数据 */
59
60     /* 获取互斥体,可以被信号打断 */
61     if (mutex_lock_interruptible(&gpioled.lock)) {
    
```

```

62     return -ERESTARTSYS;
63 }
64 #if 0
65     mutex_lock(&gpioled.lock); /* 不能被信号打断 */
66 #endif
67
68     return 0;
69 }
.....
120 static int led_release(struct inode *inode, struct file *filp)
121 {
122     struct gpioled_dev *dev = filp->private_data;
123
124     /* 释放互斥锁 */
125     mutex_unlock(&dev->lock);
126     return 0;
127 }
128
129 /* 设备操作函数 */
130 static struct file_operations gpioled_fops = {
131     .owner = THIS_MODULE,
132     .open = led_open,
133     .read = led_read,
134     .write = led_write,
135     .release = led_release,
136 };
137
138 /*
139  * @description   : 驱动出口函数
140  * @param         : 无
141  * @return        : 无
142  */
143 static int __init led_init(void)
144 {
145     int ret = 0;
146     const char *str;
147
148     /* 初始化互斥体 */
149     mutex_init(&gpioled.lock);
.....
251 }
252
253 /*

```

```

254 * @description   : 驱动出口函数
255 * @param         : 无
256 * @return        : 无
257 */
258 static void __exit led_exit(void)
259 {
260     /* 注销字符设备驱动 */
261     cdev_del(&gpioled.cdev); /* 删除 cdev */
262     unregister_chrdev_region(gpioled.devid, GPIOLED_CNT);
263     device_destroy(gpioled.class, gpioled.devid); /* 注销设备 */
264     class_destroy(gpioled.class); /* 注销类 */
265     gpio_free(gpioled.led_gpio); /* 释放 GPIO */
266 }
267
268 module_init(led_init);
269 module_exit(led_exit);
270 MODULE_LICENSE("GPL");
271 MODULE_AUTHOR("ALIENTEK");
272 MODULE_INFO(intree, "Y");
    
```

第 43 行, 定义互斥体 lock。

第 60~66 行, 在 open 函数中调用 mutex_lock_interruptible 或者 mutex_lock 获取 mutex, 成功的话就表示可以使用 LED 灯, 失败的话就会进入休眠状态, 和信号量一样。

第 125 行, 在 release 函数中调用 mutex_unlock 函数释放 mutex, 这样其他应用程序就可以获取 mutex 了。

第 149 行, 在驱动入口函数中调用 mutex_init 初始化 mutex。

互斥体和二值信号量类似, 只不过互斥体是专门用于互斥访问的。

3、编写测试 APP

测试 APP 使用 12.1.1 小节中的 atomicApp.c 即可, 将 06_atomic 中的 atomicApp.c 文件到本例程中, 并将 atomicApp.c 重命名为 mutexApp.c 即可。

12.4.2 运行测试

1、编译驱动程序

编写 Makefile 文件, 本章实验的 Makefile 文件和第五章实验基本一样, 只是将 obj-m 变量的值改为 mutex.o, Makefile 内容如下所示:

```

示例代码 12.4.2.1 Makefile 文件
1  KERNELDIR := /home/alientek/rk3568_linux_sdk/kernel
.....
4  obj-m := mutex.o
.....
11 clean:
12 $(MAKE) -C $(KERNELDIR) M=$(CURRENT_PATH) clean
    
```

第 4 行, 设置 `obj-m` 变量的值为 `mutex.o`。

输入如下命令编译出驱动模块文件:

```
make ARCH=arm64 //ARCH=arm64 必须指定, 否则编译会失败
```

编译成功以后就会生成一个名为“`mutex.ko`”的驱动模块文件。

2、编译测试 APP

输入如下命令编译测试 `mutexApp.c` 这个测试程序:

```
/opt/atk-dlrk356x-toolchain/bin/aarch64-buildroot-linux-gnu-gcc mutexApp.c -o mutexApp
```

编译成功以后就会生成 `mutexApp` 这个应用程序。

3、运行测试

在 Ubuntu 中将上一小节编译出来的 `mutex.ko` 和 `mutexApp` 这两个文件通过 `adb` 命令发送到开发板的 `/lib/modules/4.19.232` 目录下, 命令如下:

```
adb push mutex.ko mutexApp /lib/modules/4.19.232
```

发送成功以后进入到开发板目录 `lib/modules/4.19.232` 中, 输入如下命令加载 `mutex.ko` 驱动模块:

```
depmod //第一次加载驱动的时候需要运行此命令
```

```
modprobe mutex //加载驱动
```

驱动加载成功以后就可以使用 `mutexApp` 软件测试驱动是否工作正常, 测试方法和 12.3.2 中测试信号量的方法一样。

如果要卸载驱动的话输入如下命令即可:

```
rmmod mutex.ko
```

第十三章 Linux 按键输入实验

在前几章我们都是使用的 GPIO 输出功能，还没有用过 GPIO 输入功能，本章我们就来学习一下如果在 Linux 下编写 GPIO 输入驱动程序。正点原子的 ATK-DLRK3568 开发板上虽然有 5 个按键，但是这 5 个按键是 ADC 方式驱动的，所以没法用这 5 个按键来做 GPIO 输入实验。但是开发板上 JP11 这个双排排针引出了 19 个 IO，我们使用 GPIO3_C5 这个引出的 IO 来完成 GPIO 输入驱动程序，同时利用第十一章讲的原子操作来对按键值进行保护。

13.1 Linux 下按键驱动原理

按键驱动和 LED 驱动原理上来讲基本都是一样的，都是操作 GPIO，只不过一个是读取 GPIO 的高低电平，一个是从 GPIO 输出高低电平。本章我们实现按键输入，在驱动程序中使用一个整形变量来表示按键值，应用程序通过 read 函数来读取按键值，判断按键有没有按下。在这里，这个保存按键值的变量就是个共享资源，驱动程序要向其写入按键值，应用程序要读取按键值。所以我们要对其进行保护，对于整形变量而言我们首选的就是原子操作，使用原子操作对变量进行赋值以及读取。Linux 下的按键驱动原理很简单，接下来开始编写驱动。

注意，本章例程只是为了演示 Linux 下 GPIO 输入驱动的编写，实际中的按键驱动并不会采用本章中所讲解的方法，Linux 下的 input 子系统专门用于输入设备！

13.2 硬件原理图分析

开发板上的按键都是采用 ADC 驱动的，而 RK3568 的 ADC 引脚不能复用为 GPIO。所以不能直接使用按键来完成本实验。我们可以使用 JP11 这个双排排针引出的 GPIO 来完成，这里我们使用 GPIO3_C5 这个引出 IO，在开发板原理图上如图 13.2.1 所示：

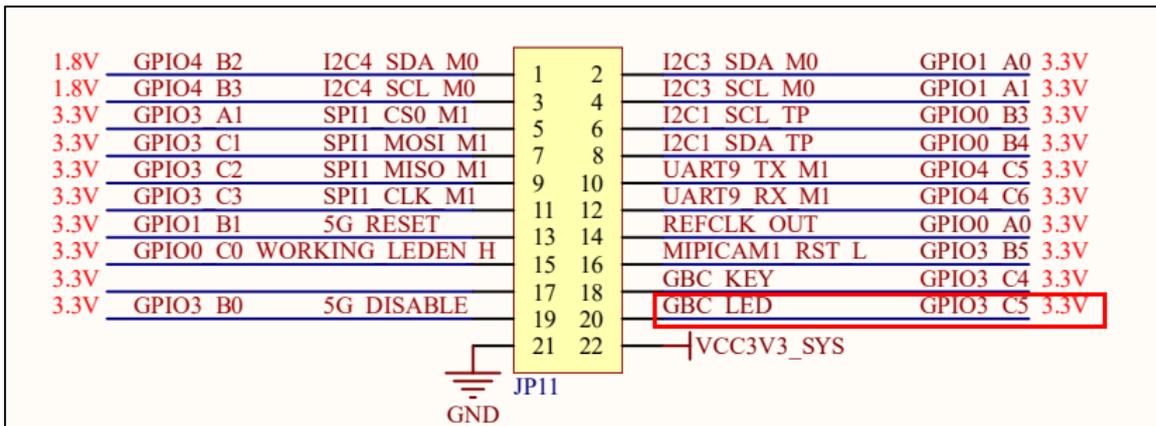


图 13.2.1 GPIO3_C5 引脚

默认情况下 GPIO3_C5 是低电平，所以我们通过使用杜邦线将图 13.2.1 中 GPIO3_C5 这个引脚接到 VCC 上的方式来模拟按键按下。也就是模拟按键按下，GPIO3_C5 为高电平，松开按键为低电平。

13.3 实验程序编写

本实验对应的例程路径为：[开发板光盘](#) → 01、[程序源码](#) → [Linux 驱动例程源码](#) → 10_key

13.3.1 修改设备树文件

1、pinctrl 设置

首先打开 rk3568-pinctrl.dtsi 文件，在 pinctrl 节点下添加 GPIO3_C5 的 pinctrl 信息，内容如下：

```

示例代码 13.3.1.1 创建 key-gpios 节点
1 key-gpios{
2     /omit-if-no-ref/
3     key_gpio: key-pin {
```

```

4     rockchip,pins =
5         <3 RK_PC5 RK_FUNC_GPIO &pcfg_pull_none>;
6     };
7 };
    
```

示例代码 13.3.1.1 是按键所用的 GPIO3_C5 这个 GPIO 的 pinctrl 信息, 也就是设置 GPIO3_C5 为 GPIO 模式, 无上下拉。

继续打开 rk3568-atk-evb1-ddr4-v10.dtsi 文件, 在根节点 “/” 下创建 KEY 节点, 节点名为 “key”, 节点内容如下:

示例代码 13.3.1.2 创建 KEY 节点

```

1 key {
2     compatible = "alientek,key";
3     pinctrl-0 = <&key_gpio>;
4     pinctrl-names = "alientek,key";
5     key-gpio = <&gpio3 RK_PC5 GPIO_ACTIVE_HIGH>;
6     status = "okay";
7 };
    
```

第 5 行, key-gpio 属性指定了 KEY 所使用的 GPIO, 高电平有效。

13.3.2 按键驱动程序编写

设备树准备好以后就可以编写驱动程序了, 新建名为 “10_key” 的文件夹, 然后在 10_key 文件夹里面创建 vscode 工程, 工作区命名为 “key”。工程创建好以后新建 key.c 文件, 在 key.c 里面输入如下内容:

示例代码 13.3.2.1 key.c 文件代码

```

1 #include <linux/types.h>
2 #include <linux/kernel.h>
3 #include <linux/delay.h>
4 #include <linux/ide.h>
5 #include <linux/init.h>
6 #include <linux/module.h>
7 #include <linux/errno.h>
8 #include <linux/gpio.h>
9 #include <linux/cdev.h>
10 #include <linux/device.h>
11 #include <linux/of.h>
12 #include <linux/of_address.h>
13 #include <linux/of_gpio.h>
14 #include <linux/semaphore.h>
15 // #include <asm/mach/map.h>
16 #include <asm/uaccess.h>
17 #include <asm/io.h>
18 /*****
19 Copyright © ALIENTEK Co., Ltd. 1998-2029. All rights reserved.
20 文件名      : key.c
    
```

```

21 作者      : 正点原子 Linux 团队
22 版本      : V1.0
23 描述      : Linux 按键输入驱动实验
24 其他      : 无
25 论坛      : www.openedv.com
26 日志      : 初版 V1.0 2022/12/23 正点原子 Linux 团队创建
27 *****/
28 #define KEY_CNT      1      /* 设备号个数 */
29 #define KEY_NAME     "key"  /* 名字 */
30
31 /* 定义按键值 */
32 #define KEY0VALUE    0XF0   /* 按键值 */
33 #define INVAKEY     0X00   /* 无效的按键值 */
34
35 /* key 设备结构体 */
36 struct key_dev{
37     dev_t devid;          /* 设备号 */
38     struct cdev cdev;    /* cdev */
39     struct class *class; /* 类 */
40     struct device *device; /* 设备 */
41     int major;           /* 主设备号 */
42     int minor;          /* 次设备号 */
43     struct device_node *nd; /* 设备节点 */
44     int key_gpio;       /* key 所使用的 GPIO 编号 */
45     atomic_t keyvalue;  /* 按键值 */
46 };
47
48 static struct key_dev keydev; /* key 设备 */
49
50 /*
51  * @description : 初始化按键 IO, open 函数打开驱动的时候初始化按键所使用的
52  *               的 GPIO 引脚。
53  * @param      : 无
54  * @return     : 无
55  */
56 static int keyio_init(void)
57 {
58     int ret;
59     const char *str;
60
61     /* 设置 LED 所使用的 GPIO */
62     /* 1、获取设备节点: keydev */
63     keydev.nd = of_find_node_by_path("/key");
    
```

```

64     if(keydev.nd == NULL) {
65         printk("keydev node not find!\r\n");
66         return -EINVAL;
67     }
68
69     /* 2.读取 status 属性 */
70     ret = of_property_read_string(keydev.nd, "status", &str);
71     if(ret < 0)
72         return -EINVAL;
73
74     if (strcmp(str, "okay"))
75         return -EINVAL;
76
77     /* 3、获取 compatible 属性值并进行匹配 */
78     ret = of_property_read_string(keydev.nd, "compatible", &str);
79     if(ret < 0) {
80         printk("keydev: Failed to get compatible property\n");
81         return -EINVAL;
82     }
83
84     if (strcmp(str, "alientek,key")) {
85         printk("keydev: Compatible match failed\n");
86         return -EINVAL;
87     }
88
89     /* 4、 获取设备树中的 gpio 属性，得到 KEY0 所使用的 KEY 编号 */
90     keydev.key_gpio = of_get_named_gpio(keydev.nd, "key-gpio", 0);
91     if(keydev.key_gpio < 0) {
92         printk("can't get key-gpio");
93         return -EINVAL;
94     }
95     printk("key-gpio num = %d\r\n", keydev.key_gpio);
96
97     /* 5.向 gpio 子系统申请使用 GPIO */
98     ret = gpio_request(keydev.key_gpio, "KEY0");
99     if (ret) {
100         printk(KERN_ERR "keydev: Failed to request key-gpio\n");
101         return ret;
102     }
103
104     /* 6、设置 GPIO3_C5 输入模式 */
105     ret = gpio_direction_input(keydev.key_gpio);
106     if(ret < 0) {
    
```

```

107     printk("can't set gpio!\r\n");
108     return ret;
109 }
110 return 0;
111 }
112
113 /*
114 * @description   : 打开设备
115 * @param - inode: 传递给驱动的 inode
116 * @param - filp  : 设备文件, file 结构体有个叫做 private_data 的成员变量
117 *                  一般在 open 的时候将 private_data 指向设备结构体。
118 * @return        : 0 成功;其他 失败
119 */
120 static int key_open(struct inode *inode, struct file *filp)
121 {
122     int ret = 0;
123     filp->private_data = &keydev; /* 设置私有数据 */
124
125     ret = keyio_init();           /* 初始化按键 IO */
126     if (ret < 0) {
127         return ret;
128     }
129
130     return 0;
131 }
132
133 /*
134 * @description   : 从设备读取数据
135 * @param - filp  : 要打开的设备文件(文件描述符)
136 * @param - buf   : 返回给用户空间的数据缓冲区
137 * @param - cnt   : 要读取的数据长度
138 * @param - offt  : 相对于文件首地址的偏移
139 * @return        : 读取的字节数, 如果为负值, 表示读取失败
140 */
141 static ssize_t key_read(struct file *filp, char __user *buf,
142                        size_t cnt, loff_t *offt)
143 {
144     int ret = 0;
145     int value;
146     struct key_dev *dev = filp->private_data;
147
148     if (gpio_get_value(dev->key_gpio) == 1) { /* key0 按下 */
149         while(gpio_get_value(dev->key_gpio)); /* 等待按键释放 */
    
```

```

149     atomic_set(&dev->keyvalue, KEY0VALUE);
150 } else {
151     atomic_set(&dev->keyvalue, INVKEY);          /* 无效的按键值 */
152 }
153
154 value = atomic_read(&dev->keyvalue);
155 ret = copy_to_user(buf, &value, sizeof(value));
156 return ret;
157 }
158
159 /*
160 * @description   : 向设备写数据
161 * @param - filp  : 设备文件, 表示打开的文件描述符
162 * @param - buf   : 要写给设备写入的数据
163 * @param - cnt   : 要写入的数据长度
164 * @param - offt  : 相对于文件首地址的偏移
165 * @return        : 写入的字节数, 如果为负值, 表示写入失败
166 */
167 static ssize_t key_write(struct file *filp, const char __user *buf,
168                          size_t cnt, loff_t *offt)
169 {
170     return 0;
171 }
172 /*
173 * @description   : 关闭/释放设备
174 * @param - filp  : 要关闭的设备文件(文件描述符)
175 * @return        : 0 成功;其他 失败
176 */
177 static int key_release(struct inode *inode, struct file *filp)
178 {
179     struct key_dev *dev = filp->private_data;
180     gpio_free(dev->key_gpio);
181
182     return 0;
183 }
184
185 /* 设备操作函数 */
186 static struct file_operations key_fops = {
187     .owner = THIS_MODULE,
188     .open = key_open,
189     .read = key_read,
190     .write = key_write,

```

```

191     .release = key_release,
192 };
193
194 /*
195  * @description   : 驱动入口函数
196  * @param         : 无
197  * @return        : 无
198  */
199 static int __init mykey_init(void)
200 {
201     int ret;
202     /* 1、初始化原子变量 */
203     keydev.keyvalue= (atomic_t)ATOMIC_INIT(0);
204
205     /* 2、原子变量初始值为 INVAKEY */
206     atomic_set(&keydev.keyvalue, INVAKEY);
207
208     /* 注册字符设备驱动 */
209     /* 1、创建设备号 */
210     if (keydev.major) { /* 定义了设备号 */
211         keydev.devid = MKDEV(keydev.major, 0);
212         ret = register_chrdev_region(keydev.devid, KEY_CNT,
213                                     KEY_NAME);
214
215         if(ret < 0) {
216             pr_err("cannot register %s char driver [ret=%d]\n",
217                   KEY_NAME, KEY_CNT);
218             return -EIO;
219         }
220     } else { /* 没有定义设备号 */
221         ret = alloc_chrdev_region(&keydev.devid, 0, KEY_CNT,
222                                 KEY_NAME); /* 申请设备号 */
223
224         if(ret < 0) {
225             pr_err("%s Couldn't alloc_chrdev_region, ret=%d\r\n",
226                   KEY_NAME, ret);
227             return -EIO;
228         }
229
230         keydev.major = MAJOR(keydev.devid); /* 获取分配号的主设备号 */
231         keydev.minor = MINOR(keydev.devid); /* 获取分配号的次设备号 */
232     }
233     printk("keydev major=%d,minor=%d\r\n",keydev.major,
234           keydev.minor);
235
236     /* 2、初始化 cdev */

```

```

229     keydev.cdev.owner = THIS_MODULE;
230     cdev_init(&keydev.cdev, &key_fops);
231
232     /* 3、添加一个 cdev */
233     cdev_add(&keydev.cdev, keydev.devid, KEY_CNT);
234     if(ret < 0)
235         goto del_unregister;
236
237     /* 4、创建类 */
238     keydev.class = class_create(THIS_MODULE, KEY_NAME);
239     if (IS_ERR(keydev.class)) {
240         goto del_cdev;
241     }
242
243     /* 5、创建设备 */
244     keydev.device = device_create(keydev.class, NULL, keydev.devid,
                                   NULL, KEY_NAME);
245     if (IS_ERR(keydev.device)) {
246         goto destroy_class;
247     }
248     return 0;
249
250 destroy_class:
251     device_destroy(keydev.class, keydev.devid);
252 del_cdev:
253     cdev_del(&keydev.cdev);
254 del_unregister:
255     unregister_chrdev_region(keydev.devid, KEY_CNT);
256     return -EIO;
257 }
258
259 /*
260 * @description   : 驱动出口函数
261 * @param         : 无
262 * @return        : 无
263 */
264 static void __exit mykey_exit(void)
265 {
266     /* 注销字符设备驱动 */
267     cdev_del(&keydev.cdev); /* 删除 cdev */
268     unregister_chrdev_region(keydev.devid, KEY_CNT); /* 注销设备号 */
269
270     device_destroy(keydev.class, keydev.devid);

```

```

271     class_destroy(keydev.class);
272 }
273
274 module_init(mykey_init);
275 module_exit(mykey_exit);
276 MODULE_LICENSE("GPL");
277 MODULE_AUTHOR("ALIEN TEK");
278 MODULE_INFO(intree, "Y");
    
```

第 36~46 行, 结构体 `key_dev` 为按键的设备结构体, 第 45 行的原子变量 `keyvalue` 用于记录按键值。

第 56~111 行, 函数 `keyio_init` 用于初始化按键, 从设备树中获取按键的 `gpio` 信息, 然后设置为输入。这里将按键的初始化代码提取出来, 将其作为独立的一个函数有利于提高程序的模块化设计。

第 120~131 行, `key_open` 函数通过调用 `keyio_init` 函数来始化按键所使用的 IO, 应用程序每次打开按键驱动文件的时候都会初始化一次按键 IO。

第 141~157 行, `key_read` 函数, 应用程序通过 `read` 函数读取按键值的时候此函数就会执行。第 147 行读取按键 IO 的电平, 如果为 1 的话就表示按键按下了, 如果按键按下的话第 148 等待按键释放。按键释放以后标记按键值为 `0XF0`。

第 199~257 行, 驱动入口函数, 第 206 行调用 `atomic_set` 函数初始化原子变量默认为无效值。

第 264~272 行, 驱动出口函数。

`key.c` 文件代码很简单, 重点就是 `key_read` 函数读取按键值, 要对 `keyvalue` 进行保护。

13.3.3 编写测试 APP

新建名为 `keyApp.c` 的文件, 然后输入如下所示内容:

示例代码 13.3.3.1 `keyApp.c` 文件代码

```

1 #include "stdio.h"
2 #include "unistd.h"
3 #include "sys/types.h"
4 #include "sys/stat.h"
5 #include "fcntl.h"
6 #include "stdlib.h"
7 #include "string.h"
8 /*****
9 Copyright © ALIENTEK Co., Ltd. 1998-2029. All rights reserved.
10 文件名      : keyApp.c
11 作者        : 正点原子 Linux 团队
12 版本        : V1.0
13 描述        : 按键输入测试应用程序
14 其他        : 无
15 使用方法    : ./keyApp /dev/key
16 论坛        : www.openedv.com
17 日志        : 初版 V1.0 2021/01/5 正点原子 Linux 团队创建
    
```

```

18 *****/
19
20 /* 定义按键值 */
21 #define KEY0VALUE      0XF0
22 #define INVAKEY       0X00
23
24 /*
25 * @description      : main 主程序
26 * @param - argc    : argv 数组元素个数
27 * @param - argv    : 具体参数
28 * @return           : 0 成功;其他 失败
29 */
30 int main(int argc, char *argv[])
31 {
32     int fd, ret;
33     char *filename;
34     int keyvalue;
35
36     if(argc != 2){
37         printf("Error Usage!\r\n");
38         return -1;
39     }
40
41     filename = argv[1];
42
43     /* 打开 key 驱动 */
44     fd = open(filename, O_RDWR);
45     if(fd < 0){
46         printf("file %s open failed!\r\n", argv[1]);
47         return -1;
48     }
49
50     /* 循环读取按键值数据! */
51     while(1) {
52         read(fd, &keyvalue, sizeof(keyvalue));
53         if (keyvalue == KEY0VALUE) { /* KEY0 */
54             printf("KEY0 Press, value = %#X\r\n", keyvalue);/* 按下 */
55         }
56     }
57
58     ret= close(fd); /* 关闭文件 */
59     if(ret < 0){
60         printf("file %s close failed!\r\n", argv[1]);

```

```

61     return -1;
62 }
63     return 0;
64 }
    
```

第 51~56 行，循环读取/dev/key 文件，也就是循环读取按键值，并且将按键值打印出来。

13.4 运行测试

13.4.1 编译驱动程序和测试 APP

1、编译驱动程序

编写 Makefile 文件，本章实验的 Makefile 文件和第五章实验基本一样，只是将 obj-m 变量的值改为 key.o，Makefile 内容如下所示：

示例代码 13.4.1.1 Makefile 文件

```

1 KERNELDIR := /home/alientek/rk3568_linux_sdk/kernel
.....
4 obj-m := key.o
.....
11 clean:
12 $(MAKE) -C $(KERNELDIR) M=$(CURRENT_PATH) clean
    
```

第 4 行，设置 obj-m 变量的值为 key.o。

输入如下命令编译出驱动模块文件：

```
make ARCH=arm64 //ARCH=arm64 必须指定，否则编译会失败
编译成功以后就会生成一个名为“key.ko”的驱动模块文件。
```

2、编译测试 APP

输入如下命令编译测试 keyApp.c 这个测试程序：

```
/opt/atk-dlrk356x-toolchain/bin/aarch64-buildroot-linux-gnu-gcc keyApp.c -o keyApp
编译成功以后就会生成 keyApp 这个应用程序。
```

13.4.2 运行测试

在 Ubuntu 中将上一小节编译出来的 key.ko 和 keyApp 这两个文件通过 adb 命令发送到开发板的/lib/modules/4.19.232 目录下，命令如下：

```
adb push key.ko keyApp /lib/modules/4.19.232
```

发送成功以后进入到开发板目录 lib/modules/4.19.232 中，输入如下命令加载 key.ko 驱动模块：

```
depmod //第一次加载驱动的时候需要运行此命令
modprobe key //加载驱动
```

驱动加载成功以后如下命令来测试：

```
./keyApp /dev/key
```

输入上述命令以后终端显示如图 13.4.2.1 所示：

```
root@ATK-DLRK356X:/lib/modules/4.19.232# ./keyApp /dev/key
[ 741.549797] key-gpio num = 117
```

图 13.4.2.1 测试 APP 运行界面

从图 13.4.2.1 可以看出, GPIO3_C5 对应的编号是 117, GPIO0 和 GPIO1 和 GPIO2 这三组每个都有 32 个 IO, GPIO3_C5 这个 IO 在 GPIO3 这一组里面是 21 号。所以 GPIO3_C5 在整个 GPIO 里面的编号就是 $32*2+21=117$ 号。

使用杜邦线将图 13.2.1 中 GPIO3_C5 这个 IO 接到开发板的 3.3V 电压上, 模拟按键被按下, keyApp 就会获取并且输出按键信息, 如图 13.4.2.2 所示:

```
root@ATK-DLRK356X:/lib/modules/4.19.232# ./keyApp /dev/key
[ 741.549797] key-gpio num = 117

KEY0 Press, value = 0XF0
```

图 13.4.2.2 按键运行结果

从图 13.4.2.2 可以看出, 当我们按下“按键”以后就会打印出“KEY0 Press, value = 0XF0”, 表示按键按下。但是大家可能会发现, 有时候按下一次“按键”但是会输出好几行“KEY0 Press, value = 0XF0”, 这是因为我们的代码没有做按键消抖处理。

如果要卸载驱动的话输入如下命令即可:

```
rmmod key.ko
```

第十四章 Linux 内核定时器实验

定时器是我们最常用到的功能，一般用来完成定时功能，本章我们就来学习一下 Linux 内核提供的定时器 API 函数，通过这些定时器 API 函数我们可以完成很多要求定时的应用。Linux 内核也提供了短延时函数，比如微秒、纳秒、毫秒延时函数，本章我们就来学习一下这些和时间有关的功能。

14.1 Linux 时间管理和内核定时器简介

14.1.1 内核时间管理简介

学习过 UCOS 或 FreeRTOS 的同学应该知道，UCOS 或 FreeRTOS 是需要一个硬件定时器提供系统时钟，一般使用 Systick 作为系统时钟源。同理，Linux 要运行，也是需要一个系统时钟的，至于这个系统时钟是由哪个定时器提供的，笔者没有去研究过 Linux 内核，但是在 Cortex-A7 内核中有个通用定时器，在《Cortex-A7 Technical ReferenceManua.pdf》的“9:Generic Timer”章节有简单的讲解，关于这个通用定时器的详细内容，可以参考《ARM ArchitectureReference Manual ARMv7-A and ARMv7-R edition.pdf》的“chapter B8 The Generic Timer”章节。这个通用定时器是可选的，按照笔者学习 FreeRTOS 和 STM32 的经验，猜测 Linux 会将这个通用定时器作为 Linux 系统时钟源(前提是 SOC 得选配这个通用定时器)。具体是怎么做的笔者没有深入研究过，这里仅仅是猜测！不过对于我们 Linux 驱动编写者来说，不需要深入研究这些具体的实现，只需要掌握相应的 API 函数即可，除非你是内核编写者或者内核爱好者。

Linux 内核中有大量的函数需要时间管理，比如周期性的调度程序、延时程序、对于我们驱动编写者来说最常用的定时器。硬件定时器提供时钟源，时钟源的频率可以设置，设置好以后就周期性的产生定时中断，系统使用定时中断来计时。中断周期性产生的频率就是系统频率，也叫做节拍率(tick rate)(有的资料也叫系统频率)，比如 100Hz、1000Hz 等等说的就是系统节拍率。系统节拍率是可以设置的，单位是 Hz，我们在编译 Linux 内核的时候可以通过图形化界面设置系统节拍率。

```
make ARCH=arm64 menuconfig // 在内核路径下执行打开配置界面
```

按照如下路径打开配置界面：

```
-> Kernel Features
```

```
-> Timer frequency (<choice> [=y])
```

选中“Timer frequency”，打开以后如图 14.1.1.1 所示：

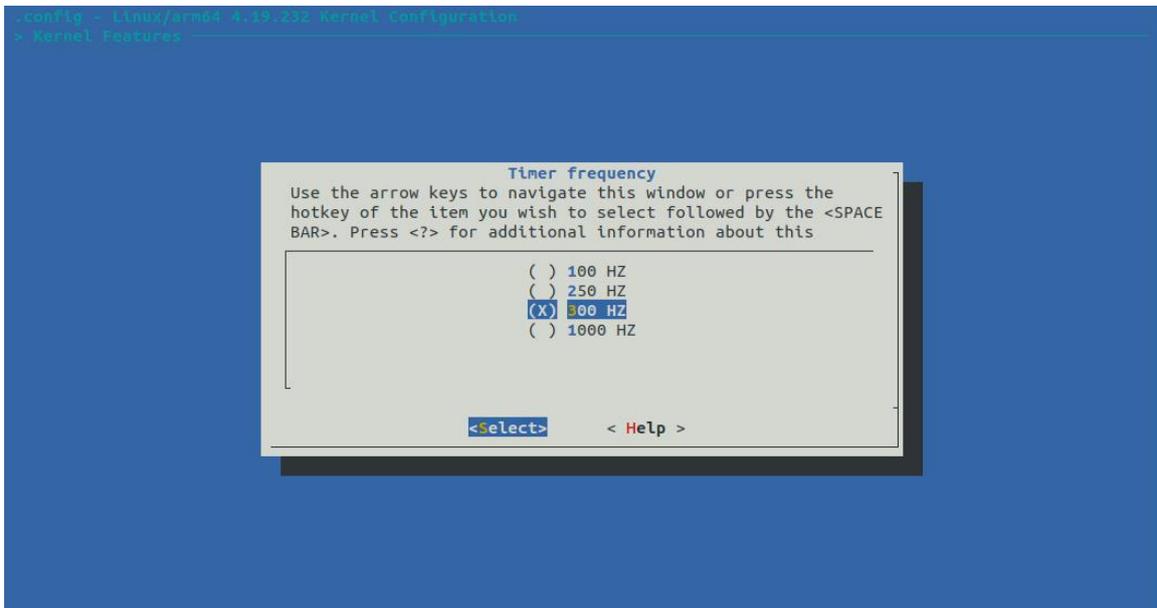


图 14.1.1.1 系统节拍率设置

从图 14.1.1.1 可以看出，可选的系统节拍率为 100Hz、200Hz、250Hz、300Hz、500Hz 和 1000Hz，默认情况下选择 300Hz。设置好以后打开 Linux 内核源码根目录下的.config 文件，在此文件中有如图 14.1.1.2 所示定义：

```

383 # CONFIG_HZ_250 is not set
384 CONFIG_HZ_300=y
385 # CONFIG_HZ_1000 is not set
386 CONFIG_HZ=300
387 CONFIG_SCHED_HRTICK=y
388 CONFIG_ARCH_SUPPORTS_DEBUG_PAGEALLOC=y
    
```

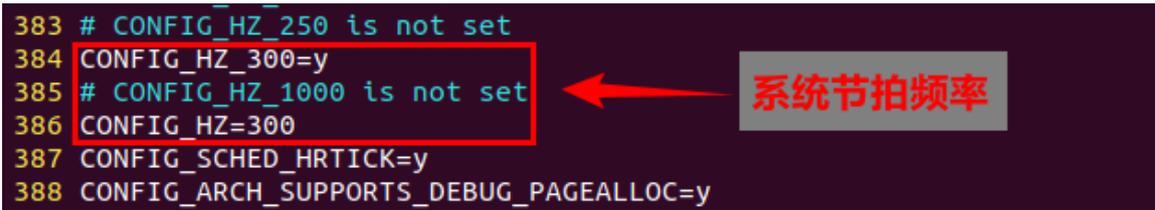


图 14.1.1.2 系统节拍率

图 14.1.1.2 中的 CONFIG_HZ 为 300, Linux 内核会使用 CONFIG_HZ 来设置自己的系统时钟。打开文件 include/asm-generic/param.h, 有如下内容:

示例代码 14.1.1.1 include/asm-generic/param.h 文件代码段

```

7 # undef HZ
8 # define HZ                CONFIG_HZ
9 # define USER_HZ           100
    
```

第 7 行定义了一个宏 HZ, 宏 HZ 就是 CONFIG_HZ, 因此 HZ=300, 我们后面编写 Linux 驱动的时候会常常用到 HZ, 因为 HZ 表示一秒的节拍数, 也就是频率。

大多数初学者看到系统节拍率默认为 300Hz 的时候都会有疑问, 怎么这么小? 甚至有的厂家会将系统节拍设置为 100Hz。为什么不选择大一点的呢? 这里就引出了一个问题: 高节拍率和低节拍率的优缺点:

①、高节拍率会提高系统时间精度, 如果采用 100Hz 的节拍率, 时间精度就是 10ms, 采用 1000Hz 的话时间精度就是 1ms, 精度提高了 10 倍。高精度时钟的好处有很多, 对于那些对时间要求严格的函数来说, 能够以更高的精度运行, 时间测量也更加准确。

②、高节拍率会导致中断的产生更加频繁, 频繁的中断会加剧系统的负担, 1000Hz 和 100Hz 的系统节拍率相比, 系统要花费 10 倍的“精力”去处理中断。中断服务函数占用处理器的时间增加, 但是现在的处理器性能都很强大, 所以采用 1000Hz 的系统节拍率并不会增加太大的负载压力。根据自己的实际情况, 选择合适的系统节拍率, 本教程我们全部采用默认的 300Hz 系统节拍率。

Linux 内核使用全局变量 jiffies 来记录系统从启动以来的系统节拍数, 系统启动的时候会将 jiffies 初始化为 0, jiffies 定义在文件 include/linux/jiffies.h 中, 定义如下:

示例代码 14.1.1.2 include/jiffies.h 文件代码段

```

80 extern u64 __cacheline_aligned_in_smp jiffies_64;
81 extern unsigned long volatile __cacheline_aligned_in_smp
__jiffy_arch_data jiffies;
    
```

第 80 行, 定义了一个 64 位的 jiffies_64。

第 81 行, 定义了一个 unsigned long 类型的 32 位的 jiffies。

jiffies_64 和 jiffies 其实是同一个东西, jiffies_64 用于 64 位系统, 而 jiffies 用于 32 位系统。为了兼容不同的硬件, jiffies 其实就是 jiffies_64 的低 32 位, jiffies_64 和 jiffies 的结构如图 14.1.1.3 所示:

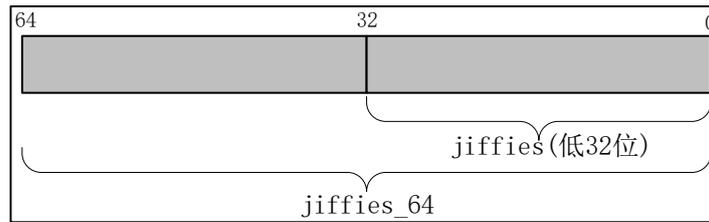


图 14.1.1.3 jiffies_64 和 jiffies 结构图

当我们访问 jiffies 的时候其实访问的是 jiffies_64 的低 32 位，使用 get_jiffies_64 这个函数可以获取 jiffies_64 的值。在 32 位的系统上读取的是 jiffies，在 64 位的系统上 jiffies 和 jiffies_64 表示同一个变量，因此也可以直接读取 jiffies 的值。所以不管是 32 位的系统还是 64 位系统，都可以使用 jiffies。

前面说了 HZ 表示每秒的节拍数，jiffies 表示系统运行的 jiffies 节拍数，所以 jiffies/HZ 就是系统运行时间，单位为秒。不管是 32 位还是 64 位的 jiffies，都有溢出的风险，溢出以后会重新从 0 开始计数，相当于绕回来了，因此有些资料也将这个现象也叫做绕回。假如 HZ 为最大值 1000 的时候，32 位的 jiffies 只需要 49.7 天就发生了绕回，对于 64 位的 jiffies 来说大概需要 5.8 亿年才能绕回，因此 jiffies_64 的绕回忽略不计。处理 32 位 jiffies 的绕回显得尤为重要，Linux 内核提供了如表 14.1.1.1 所示的几个 API 函数来处理绕回。

函数	描述
time_after(unkown, known)	unkown 通常为 jiffies，known 通常是需要对比的值。
time_before(unkown, known)	
time_after_eq(unkown, known)	
time_before_eq(unkown, known)	

表 14.1.1.1 处理绕回的 API 函数

如果 unkown 超过 known 的话，time_after 函数返回真，否则返回假。如果 unkown 没有超过 known 的话 time_before 函数返回真，否则返回假。time_after_eq 函数和 time_after 函数类似，只是多了判断等于这个条件。同理，time_before_eq 函数和 time_before 函数也类似。比如我们要判断某段代码执行时间有没有超时，此时就可以使用如下所示代码：

示例代码 14.1.1.3 使用 jiffies 判断超时

```

1 unsigned long timeout;
2 timeout = jiffies + (2 * HZ); /* 超时的时间点 */
3
4 /*****
5  具体的代码
6  *****/
7
8 /* 判断有没有超时 */
9 if(time_before(jiffies, timeout)) {
10     /* 超时未发生 */
11 } else {
12     /* 超时发生 */
13 }
    
```

timeout 就是超时时间点，比如我们要判断代码执行时间是不是超过了 2 秒，那么超时时间点就是 jiffies+(2*HZ)，如果 jiffies 大于 timeout 那就表示超时了，否则就是没有超时。第 4~6 行

就是具体的代码段。第 9 行通过函数 `time_before` 来判断 `jiffies` 是否小于 `timeout`，如果小于的话就表示没有超时。

为了方便开发，Linux 内核提供了几个 `jiffies` 和 `ms`、`us`、`ns` 之间的转换函数，如表 14.1.1.2 所示：

函数	描述
<code>int jiffies_to_msecs(const unsigned long j)</code>	将 <code>jiffies</code> 类型的参数 <code>j</code> 分别转换为对应的毫秒、微秒、纳秒。
<code>int jiffies_to_usecs(const unsigned long j)</code>	
<code>u64 jiffies_to_nsecs(const unsigned long j)</code>	
<code>long msecs_to_jiffies(const unsigned int m)</code>	将毫秒、微秒、纳秒转换为 <code>jiffies</code> 类型。
<code>long usecs_to_jiffies(const unsigned int u)</code>	
<code>unsigned long nsecs_to_jiffies(u64 n)</code>	

表 14.1.1.2 `jiffies` 和 `ms`、`us`、`ns` 之间的转换函数

14.1.2 内核定时器简介

定时器是一个很常用的功能，需要周期性处理的工作都要用到定时器，Linux 内核定时器采用系统时钟来实现。Linux 内核定时器使用很简单，只需要提供超时时间(相当于定时值)和定时处理函数即可，当超时时间到了以后设置的定时处理函数就会执行，和我们使用硬件定时器的套路一样，只是使用内核定时器不需要做一大堆的寄存器初始化工作。在使用内核定时器的时候要注意一点，内核定时器并不是周期性运行的，超时以后就会自动关闭，因此如果想要实现周期性定时，那么就需要在定时处理函数中重新开启定时器。Linux 内核使用 `timer_list` 结构体表示内核定时器，`timer_list` 定义在文件 `include/linux/timer.h` 中，定义如下：

示例代码 14.1.2.1 `timer_list` 结构体

```

11 struct timer_list {
12     /*
13      * All fields that change during normal runtime grouped to the
14      * same cacheline
15      */
16     struct hlist_node entry;
17     unsigned long expires; /* 定时器超时时间，单位是节拍数 */
18     void (*function)(struct timer_list *); /* 定时处理函数 */
19     u32 flags; /* 标志位 */
20
21 #ifdef CONFIG_LOCKDEP
22     struct lockdep_map lockdep_map;
23 #endif
24 };
    
```

要使用内核定时器首先要先定义一个 `timer_list` 变量，表示定时器，`timer_list` 结构体的 `expires` 成员变量表示超时时间，单位为节拍数。比如我们现在需要定义一个周期为 2 秒的定时器，那么这个定时器的超时时间就是 `jiffies+(2*HZ)`，因此 `expires=jiffies+(2*HZ)`。`function` 就是定时器超时以后的定时处理函数，我们要做的工作就放到这个函数里面，需要我们编写这个定时处理函数，`function` 函数的形参就是我们定义的 `timer_list` 变量。

定义好定时器以后还需要通过一系列的 API 函数来初始化此定时器，这些函数如下：

1、timer_setup 函数

timer_setup 函数负责初始化 timer_list 类型变量, 当我们定义了一个 timer_list 变量以后一定要先用 timer_setup 初始化一下。timer_setup 函数原型如下:

```
void timer_setup(struct timer_list *timer, void (*func)(struct timer_list *), unsigned int flags)
```

函数参数和返回值含义如下:

timer: 要初始化定时器。

func: 定时器的回调函数, 此函数的形参是当前定时器的变量。

flags: 标志位, 直接给 0 就行。

返回值: 没有返回值。

2、add_timer 函数

add_timer 函数用于向 Linux 内核注册定时器, 使用 add_timer 函数向内核注册定时器以后, 定时器就会开始运行, 函数原型如下:

```
void add_timer(struct timer_list *timer)
```

函数参数和返回值含义如下:

timer: 要注册的定时器。

返回值: 没有返回值。

3、del_timer 函数

del_timer 函数用于删除一个定时器, 不管定时器有没有被激活, 都可以使用此函数删除。在多处理器系统上, 定时器可能会在其他的处理器上运行, 因此在调用 del_timer 函数删除定时器之前要先等待其他处理器的定时处理器函数退出。del_timer 函数原型如下:

```
int del_timer(struct timer_list * timer)
```

函数参数和返回值含义如下:

timer: 要删除的定时器。

返回值: 0, 定时器还没被激活; 1, 定时器已经激活。

4、del_timer_sync 函数

del_timer_sync 函数是 del_timer 函数的同步版, 会等待其他处理器使用完定时器再删除, del_timer_sync 不能使用在中断上下文中。del_timer_sync 函数原型如下所示:

```
int del_timer_sync(struct timer_list *timer)
```

函数参数和返回值含义如下:

timer: 要删除的定时器。

返回值: 0, 定时器还没被激活; 1, 定时器已经激活。

5、mod_timer 函数

mod_timer 函数用于修改定时值, 如果定时器还没有激活的话, mod_timer 函数会激活定时器! 函数原型如下:

```
int mod_timer(struct timer_list *timer, unsigned long expires)
```

函数参数和返回值含义如下:

timer: 要修改超时时间(定时值)的定时器。

expires: 修改后的超时时间。

返回值: 0, 调用 mod_timer 函数前定时器未被激活; 1, 调用 mod_timer 函数前定时器已被激活。

关于内核定时器常用的 API 函数就讲这些, 内核定时器一般的使用流程如下所示:

示例代码 14.1.2.2 内核定时器使用方法演示

```

1  struct timer_list timer;    /* 定义定时器 */
2
3  /* 定时器回调函数 */
4  void function(struct timer_list *arg)
5  {
6      /*
7       * 定时器处理代码
8       */
9
10     /* 如果需要定时器周期性运行的话就使用 mod_timer
11      * 函数重新设置超时值并且启动定时器。
12      */
13     mod_timer(&dev->timertest, jiffies + msecs_to_jiffies(2000));
14 }
15
16 /* 初始化函数 */
17 void init(void)
18 {
19     timer_setup(&timerdev.timer, timer_function, 0); /* 初始化定时器 */
20     timer.expires=jiffies + msecs_to_jiffies(2000); /* 超时时间 2 秒 */
21     add_timer(&timer); /* 启动定时器 */
22 }
23
24 /* 退出函数 */
25 void exit(void)
26 {
27     del_timer(&timer); /* 删除定时器 */
28     /* 或者使用 */
29     del_timer_sync(&timer);
30 }
    
```

14.1.3 Linux 内核短延时函数

有时候我们需要在内核中实现短延时，尤其是在 Linux 驱动中。Linux 内核提供了毫秒、微秒和纳秒延时函数，这三个函数如表 14.1.3.1 所示：

函数	描述
void ndelay(unsigned long nsecs)	纳秒、微秒和毫秒延时函数。
void udelay(unsigned long usecs)	
void mdelay(unsigned long mseces)	

表 14.1.3.1 内核短延时函数

14.2 硬件原理图分析

本章使用通过设置一个定时器来实现周期性的闪烁 LED 灯，因此本章例程就使用到了一个 LED 灯，本实验的硬件原理参考 6.2 小节即可。

14.3 实验程序编写

本实验对应的例程路径为：[开发板光盘](#)→01、[程序源码](#)→Linux 驱动例程→11_timer

本章实验我们使用内核定时器周期性的点亮和熄灭开发板上的 LED 灯，LED 灯的闪烁周期由内核定时器来设置，测试应用程序可以控制内核定时器周期。

14.3.1 修改设备树文件

本章实验使用到了 LED 灯，LED 灯的设备树节点信息使用 10.4.2 小节创建的即可。

14.3.2 定时器驱动程序编写

新建名为“11_timer”的文件夹，然后在 11_timer 文件夹里面创建 vscode 工程，工作区命名为“timer”。工程创建好以后新建 timer.c 文件，在 timer.c 里面输入如下内容：

示例代码 14.3.2.1 timer.c 文件代码段

```

1  #include <linux/types.h>
2  #include <linux/kernel.h>
3  #include <linux/delay.h>
4  #include <linux/ide.h>
5  #include <linux/init.h>
6  #include <linux/module.h>
7  #include <linux/errno.h>
8  #include <linux/gpio.h>
9  #include <linux/cdev.h>
10 #include <linux/device.h>
11 #include <linux/of.h>
12 #include <linux/of_address.h>
13 #include <linux/of_gpio.h>
14 #include <linux/semaphore.h>
15 #include <linux/timer.h>
16 // #include <asm/mach/map.h>
17 #include <asm/uaccess.h>
18 #include <asm/io.h>
19 /*****
20 Copyright © ALIENTEK Co., Ltd. 1998-2029. All rights reserved.
21 文件名   : timer.c
22 作者     : 正点原子 Linux 团队
23 版本     : V1.0
24 描述     : Linux 内核定时器实验
25 其他     : 无
26 论坛     : www.openedv.com
    
```

```

27 日志      : 初版 v1.0 2022/12/24 正点原子 Linux 团队创建
28 *****/
29 #define TIMER_CNT      1          /* 设备号个数 */
30 #define TIMER_NAME     "timer"    /* 名字 */
31 #define CLOSE_CMD      (_IO(0XEF, 0x1)) /* 关闭定时器 */
32 #define OPEN_CMD       (_IO(0XEF, 0x2)) /* 打开定时器 */
33 #define SETPERIOD_CMD  (_IO(0XEF, 0x3)) /* 设置定时器周期命令 */
34 #define LEDON          1          /* 开灯 */
35 #define LEDOFF         0          /* 关灯 */
36
37 /* timer 设备结构体 */
38 struct timer_dev{
39     dev_t devid;          /* 设备号 */
40     struct cdev cdev;    /* cdev */
41     struct class *class; /* 类 */
42     struct device *device; /* 设备 */
43     int major;           /* 主设备号 */
44     int minor;           /* 次设备号 */
45     struct device_node *nd; /* 设备节点 */
46     int led_gpio;        /* key 所使用的 GPIO 编号 */
47     int timeperiod;      /* 定时周期,单位为 ms */
48     struct timer_list timer; /* 定义一个定时器 */
49     spinlock_t lock;     /* 定义自旋锁 */
50 };
51
52 struct timer_dev timerdev; /* timer 设备 */
53
54 /*
55  * @description   : 初始化 LED 灯 IO, open 函数打开驱动的时候初始化 LED 灯所
56  *                 使用的 GPIO 引脚。
57  * @param        : 无
58  * @return       : 无
59  */
60 static int led_init(void)
61 {
62     int ret;
63     const char *str;
64
65     /* 设置 LED 所使用的 GPIO */
66     /* 1、获取设备节点: timerdev */
67     timerdev.nd = of_find_node_by_path("/gpioled");
68     if(timerdev.nd == NULL) {
69         printk("timerdev node not find!\r\n");
    
```

```

70     return -EINVAL;
71 }
72
73 /* 2.读取 status 属性 */
74 ret = of_property_read_string(timerdev.nd, "status", &str);
75 if(ret < 0)
76     return -EINVAL;
77
78 if (strcmp(str, "okay"))
79     return -EINVAL;
80
81 /* 3、获取 compatible 属性值并进行匹配 */
82 ret = of_property_read_string(timerdev.nd, "compatible", &str);
83 if(ret < 0) {
84     printk("timerdev: Failed to get compatible property\n");
85     return -EINVAL;
86 }
87
88 if (strcmp(str, "alientek,led")) {
89     printk("timerdev: Compatible match failed\n");
90     return -EINVAL;
91 }
92
93 /* 4、 获取设备树中的 gpio 属性, 得到 led-gpio 所使用的 led 编号 */
94 timerdev.led_gpio = of_get_named_gpio(timerdev.nd, "led-gpio",
95                                     0);
96
97 if(timerdev.led_gpio < 0) {
98     printk("can't get led-gpio");
99     return -EINVAL;
100 }
101
102 printk("led-gpio num = %d\r\n", timerdev.led_gpio);
103
104 /* 5.向 gpio 子系统申请使用 GPIO */
105 ret = gpio_request(timerdev.led_gpio, "led");
106 if (ret) {
107     printk(KERN_ERR "timerdev: Failed to request led-gpio\n");
108     return ret;
109 }
110
111 /* 6、设置 PIO 为输出, 并且输出高电平, 默认关闭 LED 灯 */
112 ret = gpio_direction_output(timerdev.led_gpio, 0);
113 if(ret < 0) {
114     printk("can't set gpio!\r\n");

```

```

112     return ret;
113 }
114 return 0;
115 }
116
117 /*
118 * @description      : 打开设备
119 * @param - inode    : 传递给驱动的 inode
120 * @param - filp     : 设备文件, file 结构体有个叫做 private_data 的成员变
121 *                   : 量一般在 open 的时候将 private_data 指向设备结构体。
122 * @return           : 0 成功;其他 失败
123 */
124 static int timer_open(struct inode *inode, struct file *filp)
125 {
126     int ret = 0;
127     filp->private_data = &timerdev; /* 设置私有数据 */
128
129     timerdev.timeperiod = 1000;    /* 默认周期为 1s */
130     ret = led_init();              /* 初始化 LED IO */
131     if (ret < 0) {
132         return ret;
133     }
134
135     return 0;
136 }
137
138 /*
139 * @description      : ioctl 函数,
140 * @param - filp    : 要打开的设备文件(文件描述符)
141 * @param - cmd     : 应用程序发送过来的命令
142 * @param - arg     : 参数
143 * @return           : 0 成功;其他 失败
144 */
145 static long timer_unlocked_ioctl(struct file *filp,
146                                 unsigned int cmd, unsigned long arg)
147 {
148     struct timer_dev *dev = (struct timer_dev *)filp->private_data;
149     int timerperiod;
150     unsigned long flags;
151
152     switch (cmd) {
153         case CLOSE_CMD: /* 关闭定时器 */
154             del_timer_sync(&dev->timer);

```

```

154         break;
155     case OPEN_CMD:      /* 打开定时器 */
156         spin_lock_irqsave(&dev->lock, flags);
157         timerperiod = dev->timeperiod;
158         spin_unlock_irqrestore(&dev->lock, flags);
159         mod_timer(&dev->timer, jiffies +
160                 msec_to_jiffies(timerperiod));
161         break;
162     case SETPERIOD_CMD: /* 设置定时器周期 */
163         spin_lock_irqsave(&dev->lock, flags);
164         dev->timeperiod = arg;
165         spin_unlock_irqrestore(&dev->lock, flags);
166         mod_timer(&dev->timer, jiffies + msec_to_jiffies(arg));
167         break;
168     default:
169         break;
170 }
171 return 0;
172 }
173 /*
174  * @description   : 关闭/释放设备
175  * @param - filp  : 要关闭的设备文件(文件描述符)
176  * @return        : 0 成功;其他 失败
177  */
178 static int led_release(struct inode *inode, struct file *filp)
179 {
180     struct timer_dev *dev = filp->private_data;
181     gpio_set_value(dev->led_gpio, 0); /* APP 结束的时候关闭 LED */
182     gpio_free(dev->led_gpio);         /* 释放 LED */
183     del_timer_sync(&dev->timer);     /* 关闭定时器 */
184
185     return 0;
186 }
187
188 /* 设备操作函数 */
189 static struct file_operations timer_fops = {
190     .owner = THIS_MODULE,
191     .open = timer_open,
192     .unlocked_ioctl = timer_unlocked_ioctl,
193     .release = led_release,
194 };
195

```

```

196 /* 定时器回调函数 */
197 void timer_function(struct timer_list *arg)
198 {
199     /* from_timer 是个宏, 可以根据结构体的成员地址, 获取到这个结构体的首地址。
200        第一个参数表示结构体, 第二个参数表示第一个参数里的一个成员, 第三个参数表
201        示第二个参数的类型, 得到第一个参数的首地址。
202
203        */
204     struct timer_dev *dev = from_timer(dev, arg, timer);
205     static int sta = 1;
206     int timerperiod;
207     unsigned long flags;
208
209     sta = !sta;    /* 每次都取反, 实现 LED 灯反转 */
210     gpio_set_value(dev->led_gpio, sta);
211
212     /* 重启定时器 */
213     spin_lock_irqsave(&dev->lock, flags);
214     timerperiod = dev->timeperiod;
215     spin_unlock_irqrestore(&dev->lock, flags);
216     mod_timer(&dev->timer, jiffies +
217              msecs_to_jiffies(dev->timeperiod));
218 }
219
220 /*
221 * @description   : 驱动入口函数
222 * @param         : 无
223 * @return        : 无
224 */
225 static int __init timer_init(void)
226 {
227     int ret;
228
229     /* 初始化自旋锁 */
230     spin_lock_init(&timerdev.lock);
231
232     /* 注册字符设备驱动 */
233     /* 1、创建设备号 */
234     if (timerdev.major) {    /* 定义了设备号 */
235         timerdev.devid = MKDEV(timerdev.major, 0);
236         ret = register_chrdev_region(timerdev.devid, TIMER_CNT,
237                                     TIMER_NAME);
238     }
239     if (ret < 0) {
240         pr_err("cannot register %s char driver [ret=%d]\n",

```

```

        TIMER_NAME, TIMER_CNT);
236     return -EIO;
237 }
238 } else {                               /* 没有定义设备号 */
239     ret = alloc_chrdev_region(&timerdev.devid, 0, TIMER_CNT,
                                TIMER_NAME); /* 申请设备号 */
240     if(ret < 0) {
241         pr_err("%s Couldn't alloc_chrdev_region, ret=%d\r\n",
                TIMER_NAME, ret);
242         return -EIO;
243     }
244     timerdev.major = MAJOR(timerdev.devid);
245     timerdev.minor = MINOR(timerdev.devid);
246 }
247 printk("timerdev major=%d,minor=%d\r\n",timerdev.major,
        timerdev.minor);
248
249 /* 2、初始化 cdev */
250 timerdev.cdev.owner = THIS_MODULE;
251 cdev_init(&timerdev.cdev, &timer_fops);
252
253 /* 3、添加一个 cdev */
254 cdev_add(&timerdev.cdev, timerdev.devid, TIMER_CNT);
255 if(ret < 0)
256     goto del_unregister;
257
258 /* 4、创建类 */
259 timerdev.class = class_create(THIS_MODULE, TIMER_NAME);
260 if (IS_ERR(timerdev.class)) {
261     goto del_cdev;
262 }
263
264 /* 5、创建设备 */
265 timerdev.device = device_create(timerdev.class, NULL,
                                timerdev.devid, NULL, TIMER_NAME);
266 if (IS_ERR(timerdev.device)) {
267     goto destroy_class;
268 }
269
270 /* 6、初始化 timer, 设置定时器处理函数, 还未设置周期, 所有不会激活定时器 */
271 timer_setup(&timerdev.timer, timer_function, 0);
272
273 return 0;
    
```

```

274
275 destroy_class:
276     device_destroy(timerdev.class, timerdev.devid);
277 del_cdev:
278     cdev_del(&timerdev.cdev);
279 del_unregister:
280     unregister_chrdev_region(timerdev.devid, TIMER_CNT);
281     return -EIO;
282 }
283
284 /*
285  * @description   : 驱动出口函数
286  * @param         : 无
287  * @return        : 无
288  */
289 static void __exit timer_exit(void)
290 {
291     del_timer_sync(&timerdev.timer);          /* 删除 timer */
292 #if 0
293     del_timer(&timerdev.tiemr);
294 #endif
295
296     /* 注销字符设备驱动 */
297     cdev_del(&timerdev.cdev); /* 删除 cdev */
298     unregister_chrdev_region(timerdev.devid, TIMER_CNT);
299
300     device_destroy(timerdev.class, timerdev.devid);
301     class_destroy(timerdev.class);
302 }
303
304 module_init(timer_init);
305 module_exit(timer_exit);
306 MODULE_LICENSE("GPL");
307 MODULE_AUTHOR("ALIEN TEK");
308 MODULE_INFO(intree, "Y");
    
```

第 38~50 行, 定时器设备结构体, 在 48 行定义了一个定时器成员变量 `timer`。

第 60~115 行, LED 灯初始化函数, 从设备树中获取 LED 灯信息, 然后初始化相应的 IO。

第 124~136 行, 函数 `timer_open`, 对应应用程序的 `open` 函数, 应用程序调用 `open` 函数打开 `/dev/timer` 驱动文件的时候此函数就会执行。此函数设置文件私有数据为 `timerdev`, 并且初始化定时周期默认为 1 秒, 最后调用 `led_init` 函数初始化 LED 所使用的 IO。

第 145~171 行, 函数 `timer_unlocked_ioctl`, 对应应用程序的 `ioctl` 函数, 应用程序调用 `ioctl` 函数向驱动发送控制信息, 此函数响应并执行。此函数有三个参数: `filp`, `cmd` 和 `arg`, 其中 `filp`

是对应的设备文件, `cmd` 是应用程序发送过来的命令信息, `arg` 是应用程序发送过来的参数, 在本章例程中 `arg` 参数表示定时周期。

一共有三种命令 `CLOSE_CMD`, `OPEN_CMD` 和 `SETPERIOD_CMD`, 这三个命令分别为关闭定时器、打开定时器、设置定时周期。这三个命令的左右如下:

CLOSE_CMD: 关闭定时器命令, 调用 `del_timer_sync` 函数关闭定时器。

OPEN_CMD: 打开定时器命令, 调用 `mod_timer` 函数打开定时器, 定时周期为 `timerdev` 的 `timeperiod` 成员变量, 定时周期默认是 1 秒。

SETPERIOD_CMD: 设置定时器周期命令, 参数 `arg` 就是新的定时周期, 设置 `timerdev` 的 `timeperiod` 成员变量为 `arg` 所表示定时周期值。并且使用 `mod_timer` 重新打开定时器, 使定时器以新的周期运行。

第 178~186 行, 函数 `timer_release`, 对应应用程序的 `close` 函数。退出的时候, 关闭 LED 和关闭定时器。

第 189~194 行, 定时器驱动操作函数集 `timer_fops`。

第 197~215 行, 函数 `timer_function`, 定时器服务函数。重点来看一下第 202 行, 这一行需要得到第 52 行定义的 `timerdev` 变量。这里使用 `from_timer` 函数来根据 `arg` 参数反推出 `timerdev` 变量地址, `from_timer` 函数其实是一个宏, 是对 `container_of` 函数的一个简单封装。`container_of` 函数的作用就是: 给定结构体中的某个成员变量的地址、该结构体类型和该成员的名字来得到这个成员所在结构体变量的首地址。比如 `timerdev` 这个结构体变量, 类型为 `timer_dev`, 而 `timer_dev` 中有个成员变量 `timer`, `timer` 是 `timer_list` 类型。因此当我们知道了 `timer` 这个成员变量的具体地址以后, 就可以根据 `container_of` 函数来反推出 `timer` 这个成员变量所属的 `timer_dev` 结构体变量首地址, 在这里就是得到 `timerdev` 地址。最后在 `timer_function` 函数中将 LED 灯的状态取反, 实现 LED 灯闪烁的效果。因为内核定时器不是循环的定时器, 执行一次以后就结束了, 因此在 214 行又调用了 `mod_timer` 函数重新开启定时器。

第 222~282 行, 函数 `timer_init`, 驱动入口函数。在第 271 行初始化定时器, 设置定时器的定时处理函数为 `timer_function`, 标志位为 0。在此函数中并没有调用 `timer_add` 函数来开启定时器, 因此定时器默认是关闭的, 除非应用程序发送打开命令。

第 289~302 行, 驱动出口函数。第 291 行调用 `del_timer_sync` 函数删除定时器, 也可以使用 `del_timer` 函数。

14.3.3 编写测试 APP

测试 APP 我们要实现的内容如下:

①、运行 APP 以后提示我们输入要测试的命令, 输入 1 表示关闭定时器、输入 2 表示打开定时器, 输入 3 设置定时器周期。

②、如果要设置定时器周期的话, 需要让用户输入要设置的周期值, 单位为毫秒。

新建名为 `timerApp.c` 的文件, 然后输入如下所示内容:

示例代码 14.3.3.1 `timerApp.c` 文件代码段

```

1  #include "stdio.h"
2  #include "unistd.h"
3  #include "sys/types.h"
4  #include "sys/stat.h"
5  #include "fcntl.h"
6  #include "stdlib.h"
7  #include "string.h"
    
```

```

8  #include <sys/ioctl.h>
9  /*****
10 Copyright © ALIENTEK Co., Ltd. 1998-2029. All rights reserved.
11 文件名      : timerApp.c
12 作者        : 正点原子 Linux 团队
13 版本        : V1.0
14 描述        : 定时器测试应用程序
15 其他        : 无
16 使用方法    : ./timertest /dev/timer 打开测试 App
17 论坛        : www.openedv.com
18 日志        : 初版 V1.0 2022/12/28 正点原子 Linux 团队创建
19 *****/
20
21 /* 命令值 */
22 #define CLOSE_CMD      (_IO(0XEF, 0x1))  /* 关闭定时器      */
23 #define OPEN_CMD       (_IO(0XEF, 0x2))  /* 打开定时器      */
24 #define SETPERIOD_CMD  (_IO(0XEF, 0x3))  /* 设置定时器周期命令 */
25
26 /*
27  * @description   : main 主程序
28  * @param - argc  : argv 数组元素个数
29  * @param - argv  : 具体参数
30  * @return        : 0 成功;其他 失败
31  */
32 int main(int argc, char *argv[])
33 {
34     int fd, ret;
35     char *filename;
36     unsigned int cmd;
37     unsigned int arg;
38     unsigned char str[100];
39
40     if (argc != 2) {
41         printf("Error Usage!\r\n");
42         return -1;
43     }
44
45     filename = argv[1];
46
47     fd = open(filename, O_RDWR);
48     if (fd < 0) {
49         printf("Can't open file %s\r\n", filename);
50         return -1;

```

```

51     }
52
53     while (1) {
54         printf("Input CMD:");
55         ret = scanf("%d", &cmd);
56         if (ret != 1) {                /* 参数输入错误 */
57             fgets(str, sizeof(str), stdin);        /* 防止卡死 */
58         }
59         if(4 == cmd)                   /* 退出 APP    */
60             goto out;
61         if(cmd == 1)                   /* 关闭 LED 灯 */
62             cmd = CLOSE_CMD;
63         else if(cmd == 2)              /* 打开 LED 灯 */
64             cmd = OPEN_CMD;
65         else if(cmd == 3) {
66             cmd = SETPERIOD_CMD;       /* 设置周期值 */
67             printf("Input Timer Period:");
68             ret = scanf("%d", &arg);
69             if (ret != 1) {            /* 参数输入错误 */
70                 fgets(str, sizeof(str), stdin);    /* 防止卡死 */
71             }
72         }
73         ioctl(fd, cmd, arg);           /* 控制定时器的打开和关闭 */
74     }
75
76 out:
77     close(fd);
78 }
    
```

第 22~24 行, 命令值。

第 53~73 行, while(1) 循环, 让用户输入要测试的命令, 然后通过第 73 行的 ioctl 函数发送给驱动程序。如果是设置定时器周期命令 SETPERIOD_CMD, 那么 ioctl 函数的 arg 参数就是用户输入的周期值。

第 59~60 行, 输入 4 就能退出 APP。

14.4 运行测试

14.4.1 编译驱动程序和测试 APP

1、编译驱动程序

编写 Makefile 文件, 本章实验的 Makefile 文件和第五章实验基本一样, 只是将 obj-m 变量的值改为 timer.o, Makefile 内容如下所示:

示例代码 14.4.1.1 Makefile 文件

```

1 KERNELDIR := /home/alientek/rk3568_linux_sdk/kernel
.....
    
```

```
4 obj-m := timer.o
.....
11 clean:
12 $(MAKE) -C $(KERNELDIR) M=$(CURRENT_PATH) clean
```

第 4 行，设置 obj-m 变量的值为 timer.o。

输入如下命令编译出驱动模块文件：

```
make ARCH=arm64 //ARCH=arm64 必须指定，否则编译会失败
```

编译成功以后就会生成一个名为“timer.ko”的驱动模块文件。

2、编译测试 APP

输入如下命令编译测试 timerApp.c 这个测试程序：

```
/opt/atk-dlrk356x-toolchain/bin/aarch64-buildroot-linux-gnu-gcc timerApp.c -o timerApp
```

编译成功以后就会生成 timerApp 这个应用程序。

14.4.2 运行测试

在 Ubuntu 中将上一小节编译出来的 timer.ko 和 timerApp 这两个文件通过 adb 命令发送到开发板的/lib/modules/4.19.232 目录下，命令如下：

```
adb push timer.ko timerApp /lib/modules/4.19.232
```

发送成功以后进入到开发板目录 lib/modules/4.19.232 中，输入如下命令加载 timer.ko 驱动模块：

```
depmod //第一次加载驱动的时候需要运行此命令
```

```
modprobe timer //加载驱动
```

驱动加载成功以后如下命令来测试：

```
./timerApp /dev/timer
```

输入上述命令以后终端提示输入命令，如图 14.4.2.1 所示：

```
root@ATK-DLRK356X:/lib/modules/4.19.232# ./timerApp /dev/timer
[ 2889.968780] led-gpio num = 16
Input CMD:█
```

图 14.4.2.1 输入命令

输入“2”，打开定时器，此时 LED 就会以默认的 1 秒周期开始闪烁。在输入“3”来设置定时周期，根据提示输入要设置的周期值，如图 14.4.2.2 所示：

```
Input CMD:3
Input Timer Period:█
```

图 14.4.2.2 设置周期值

输入“500”，表示设置定时器周期值为 500ms，设置好以后 LED 灯就会以 500ms 为间隔，开始闪烁。最后可以通过输入“1”来关闭定时器，如果要卸载驱动的话输入如下命令即可：

```
rmmod timer.ko
```

第十五章 Linux 中断实验

不管是单片机还是 Linux 下的驱动实验，中断都是频繁使用的功能，在单片机中使用中断我们需要做一大堆的工作，比如配置寄存器，使能 IRQ 等等。但是 Linux 内核提供了完善的中断框架，我们只需要申请中断，然后注册中断处理函数即可，使用非常方便，不需要一系列复杂的寄存器配置。本章我们就来学习一下如何在 Linux 下使用中断。

15.1 Linux 中断简介

15.1.1 Linux 中断 API 函数

先来回顾一下我们在做单片机开发的时候中断的处理方法:

- ①、使能中断, 初始化相应的寄存器。
- ②、编写中断服务函数, 中断发生以后相应的中断服务函数就会执行。

在 Linux 内核中也提供了大量的中断相关的 API 函数, 我们来看一下这些跟中断有关的 API 函数:

1、中断号

每个中断都有一个中断号, 通过中断号即可区分不同的中断, 有的资料也把中断号叫做中断线, 在 Linux 内核中使用一个 int 变量表示中断号。

2、request_irq 函数

在 Linux 内核中要想使用某个中断是需要申请的, request_irq 函数用于申请中断, request_irq 函数可能会导致睡眠, 因此不能在中断上下文或者其他禁止睡眠的代码段中使用 request_irq 函数。request_irq 函数会激活(使能)中断, 所以不需要我们手动去使能中断, request_irq 函数原型如下:

```
int request_irq(unsigned int    irq,
                irq_handler_t  handler,
                unsigned long   flags,
                const char     *name,
                void            *dev)
```

函数参数和返回值含义如下:

irq: 要申请中断的中断号。

handler: 中断处理函数, 当中断发生以后就会执行此中断处理函数。

flags: 中断标志, 可以在文件 include/linux/interrupt.h 里面查看所有的中断标志, 这里我们介绍几个常用的中断标志, 如表 15.1.1.1 所示:

标志	描述
IRQF_SHARED	多个设备共享一个中断线, 共享的所有中断都必须指定此标志。如果使用共享中断的话, request_irq 函数的 dev 参数就是唯一区分他们的标志。
IRQF_ONESHOT	单次中断, 中断执行一次就结束。
IRQF_TRIGGER_NONE	无触发。
IRQF_TRIGGER_RISING	上升沿触发。
IRQF_TRIGGER_FALLING	下降沿触发。
IRQF_TRIGGER_HIGH	高电平触发。
IRQF_TRIGGER_LOW	低电平触发。

表 15.1.1.1 常用的中断标志

比如在《第十三章 Linux 按键输入实验》中使用 GPIO3_C5 这个 IO, 通过将此 IO 接到 3.3V 模拟按键按下, 也就是按下以后为高电平, 因此可以设置为上升沿触发, 也就是将 flags 设置为 IRQF_TRIGGER_RISING。表 15.1.1.1 中的这些标志可以通过 “|” 来实现多种组合。

name: 中断名字, 设置以后可以在 /proc/interrupts 文件中看到对应的中断名字。

dev: 如果将 flags 设置为 IRQF_SHARED 的话, dev 用来区分不同的中断, 一般情况下将 dev 设置为设备结构体, dev 会传递给中断处理函数 irq_handler_t 的第二个参数。

返回值: 0 中断申请成功, 其他负值 中断申请失败, 如果返回-EBUSY 的话表示中断已经被申请了。

3、free_irq 函数

使用中断的时候需要通过 request_irq 函数申请, 使用完成以后就要通过 free_irq 函数释放掉相应的中断。如果中断不是共享的, 那么 free_irq 会删除中断处理函数并且禁止中断。free_irq 函数原型如下所示:

```
void free_irq(unsigned int  irq,
              void          *dev_id)
```

函数参数和返回值含义如下:

irq: 要释放的中断。

dev_id: 如果中断设置为共享(IRQF_SHARED)的话, 此参数用来区分具体的中断。共享中断只有在释放最后中断处理函数的时候才会被禁止掉。

返回值: 无。

4、中断处理函数

使用 request_irq 函数申请中断的时候需要设置中断处理函数, 中断处理函数格式如下所示:

```
irqreturn_t (*irq_handler_t)(int, void *)
```

第一个参数是要中断处理函数要相应的中断号。第二个参数是一个指向 void 的指针, 也就是个通用指针, 需要与 request_irq 函数的 dev_id 参数保持一致。用于区分共享中断的不同设备, dev_id 也可以指向设备数据结构。中断处理函数的返回值为 irqreturn_t 类型, irqreturn_t 类型定义如下所示:

示例代码 15.1.1.1 irqreturn_t 结构

```
11 enum irqreturn {
12     IRQ_NONE           = (0 << 0),
13     IRQ_HANDLED       = (1 << 0),
14     IRQ_WAKE_THREAD   = (1 << 1),
15 };
16
17 typedef enum irqreturn irqreturn_t;
```

可以看出 irqreturn_t 是个枚举类型, 一共有三种返回值。一般中断服务函数返回值使用如下形式:

```
return IRQ_RETVAL(IRQ_HANDLED)
```

5、中断使能与禁止函数

常用的中断使用和禁止函数如下所示:

```
void enable_irq(unsigned int irq)
```

```
void disable_irq(unsigned int irq)
```

enable_irq 和 disable_irq 用于使能和禁止指定的中断, irq 就是要禁止的中断号。disable_irq 函数要等到当前正在执行的中断处理函数执行完才返回, 因此使用者需要保证不会产生新的中断, 并且确保所有已经开始执行的中断处理程序已经全部退出。在这种情况下, 可以使用另外一个中断禁止函数:

```
void disable_irq_nosync(unsigned int irq)
```

`disable_irq_nosync` 函数调用以后立即返回, 不会等待当前中断处理程序执行完毕。上面三个函数都是使能或者禁止某一个中断, 有时候我们需要关闭当前处理器的整个中断系统, 也就是在学习 STM32 单片机的时候常说的关闭全局中断, 这个时候可以使用如下两个函数:

```
local_irq_enable()
local_irq_disable()
```

`local_irq_enable` 用于使能当前处理器中断系统, `local_irq_disable` 用于禁止当前处理器中断系统。假如 A 任务调用 `local_irq_disable` 关闭全局中断 10S, 当关闭了 2S 的时候 B 任务开始运行, B 任务也调用 `local_irq_disable` 关闭全局中断 3S, 3 秒以后 B 任务调用 `local_irq_enable` 函数将全局中断打开了。此时才过去 2+3=5 秒的时间, 然后全局中断就被打开了, 此时 A 任务要关闭 10S 全局中断的愿望就破灭了, 然后 A 任务就“生气了”, 结果很严重, 可能系统都要被 A 任务整崩溃。为了解决这个问题, B 任务不能直接简单粗暴的通过 `local_irq_enable` 函数来打开全局中断, 而是将中断状态恢复到以前的状态, 要考虑到别的任务的感受, 此时就要用到下面两个函数:

```
local_irq_save(flags)
local_irq_restore(flags)
```

这两个函数是一对, `local_irq_save` 函数用于禁止中断, 并且将中断状态保存在 `flags` 中。`local_irq_restore` 用于恢复中断, 将中断到 `flags` 状态。

15.1.2 上半部与下半部

在有些资料中也将上半部和下半部称为顶半部和底半部, 都是一个意思。我们在使用 `request_irq` 申请中断的时候注册的中断服务函数属于中断处理的上半部, 只要中断触发, 那么中断处理函数就会执行。我们都知道中断处理函数一定要快点执行完毕, 越短越好, 但是现实往往是残酷的, 有些中断处理过程就是比较费时间, 我们必须要对其进行处理, 缩小中断处理函数的执行时间。比如电容触摸屏通过中断通知 SOC 有触摸事件发生, SOC 响应中断, 然后通过 IIC 接口读取触摸坐标值并将其上报给系统。但是我们都知 IIC 的速度最高也只有 400Kbit/S, 所以在中断中通过 IIC 读取数据就会浪费时间。我们可以将通过 IIC 读取触摸数据的操作暂后执行, 中断处理函数仅仅相应中断, 然后清除中断标志位即可。这个时候中断处理过程就分为了两部分:

上半部: 上半部就是中断处理函数, 那些处理过程比较快, 不会占用很长时间的就可以放在上半部完成。

下半部: 如果中断处理过程比较耗时, 那么就将这些比较耗时的代码提出来, 交给下半部去执行, 这样中断处理函数就会快进快出。

因此, Linux 内核将中断分为上半部和下半部的主要目的就是实现中断处理函数的快进快出, 那些对时间敏感、执行速度快的操作可以放到中断处理函数中, 也就是上半部。剩下的所有工作都可以放到下半部去执行, 比如在上半部将数据拷贝到内存中, 关于数据的具体处理就可以放到下半部去执行。至于哪些代码属于上半部, 哪些代码属于下半部并没有明确的规定, 一切根据实际使用情况去判断, 这个就很考验驱动编写人员的功底了。这里有一些可以借鉴的参考点:

- ①、如果要处理的内容不希望被其他中断打断, 那么可以放到上半部。
- ②、如果要处理的任任务对时间敏感, 可以放到上半部。
- ③、如果要处理的任任务与硬件有关, 可以放到上半部
- ④、除了上述三点以外的其他任任务, 优先考虑放到下半部。

上半部处理很简单，直接编写中断处理函数就行了，关键是下半部该怎么做呢？Linux 内核提供了多种下半部机制，接下来我们来学习一下这些下半部机制。

1、软中断

一开始 Linux 内核提供了“bottom half”机制来实现下半部，简称“BH”。后面引入了软中断和 tasklet 来替代“BH”机制，完全可以使用软中断和 tasklet 来替代 BH，从 2.5 版本的 Linux 内核开始 BH 已经被抛弃了。Linux 内核使用结构体 `softirq_action` 表示软中断，`softirq_action` 结构体定义在文件 `include/linux/interrupt.h` 中，内容如下：

示例代码 15.1.2.1 `softirq_action` 结构体

```
491 struct softirq_action
492 {
493     void    (*action)(struct softirq_action *);
494 };
```

在 `kernel/softirq.c` 文件中一共定义了 10 个软中断，如下所示：

示例代码 15.1.2.2 `softirq_vec` 数组

```
static struct softirq_action softirq_vec[NR_SOFTIRQS];
```

`NR_SOFTIRQS` 是枚举类型，定义在文件 `include/linux/interrupt.h` 中，定义如下：

示例代码 15.1.2.3 `softirq_vec` 数组

```
enum
{
    HI_SOFTIRQ=0,           /* 高优先级软中断 */
    TIMER_SOFTIRQ,        /* 定时器软中断 */
    NET_TX_SOFTIRQ,       /* 网络数据发送软中断 */
    NET_RX_SOFTIRQ,       /* 网络数据接收软中断 */
    BLOCK_SOFTIRQ,
    IRQ_POLL_SOFTIRQ,
    TASKLET_SOFTIRQ,      /* tasklet 软中断 */
    SCHED_SOFTIRQ,       /* 调度软中断 */
    HRTIMER_SOFTIRQ,     /* 高精度定时器软中断 */
    RCU_SOFTIRQ,         /* RCU 软中断 */

    NR_SOFTIRQS
};
```

可以看出，一共有 10 个软中断，因此 `NR_SOFTIRQS` 为 10，因此数组 `softirq_vec` 有 10 个元素。`softirq_action` 结构体中的 `action` 成员变量就是软中断的服务函数，数组 `softirq_vec` 是个全局数组，因此所有的 CPU(对于 SMP 系统而言)都可以访问到，每个 CPU 都有自己的触发和控制机制，并且只执行自己所触发的软中断。但是各个 CPU 所执行的软中断服务函数确是相同的，都是数组 `softirq_vec` 中定义的 `action` 函数。要使用软中断，必须先使用 `open_softirq` 函数注册对应的软中断处理函数，`open_softirq` 函数原型如下：

```
void open_softirq(int nr, void (*action)(struct softirq_action *))
```

函数参数和返回值含义如下：

nr: 要开启的软中断，在示例代码 15.1.2.3 中选择要开启的软中断。

action: 软中断对应的处理函数。

返回值: 没有返回值。

注册好软中断以后需要通过 `raise_softirq` 函数触发, `raise_softirq` 函数原型如下:

```
void raise_softirq(unsigned int nr)
```

函数参数和返回值含义如下:

nr: 要触发的软中断, 在示例代码 15.1.2.3 中选择要注册的软中断。

返回值: 没有返回值。

软中断必须在编译的时候静态注册! Linux 内核使用 `softirq_init` 函数初始化软中断, `softirq_init` 函数定义在 `kernel/softirq.c` 文件里面, 函数内容如下:

示例代码 15.1.2.4 `softirq_init` 函数内容

```
625 void __init softirq_init(void)
626 {
627     int cpu;
628
629     for_each_possible_cpu(cpu) {
630         per_cpu(tasklet_vec, cpu).tail =
631             &per_cpu(tasklet_vec, cpu).head;
632         per_cpu(tasklet_hi_vec, cpu).tail =
633             &per_cpu(tasklet_hi_vec, cpu).head;
634     }
635
636     open_softirq(TASKLET_SOFTIRQ, tasklet_action);
637     open_softirq(HI_SOFTIRQ, tasklet_hi_action);
638 }
```

从示例代码 15.1.2.4 可以看出, `softirq_init` 函数默认会打开 `TASKLET_SOFTIRQ` 和 `HI_SOFTIRQ`。

2、tasklet

`tasklet` 是利用软中断来实现的另外一种下半部机制, 在软中断和 `tasklet` 之间, 建议大家使用 `tasklet`。`tasklet_struct` 结构体如下所示:

示例代码 15.1.2.5 `tasklet_struct` 结构体

```
542 struct tasklet_struct
543 {
544     struct tasklet_struct *next; /* 下一个 tasklet */
545     unsigned long state; /* tasklet 状态 */
546     atomic_t count; /* 计数器, 记录对 tasklet 的引用数 */
547     void (*func)(unsigned long); /* tasklet 执行的函数 */
548     unsigned long data; /* 函数 func 的参数 */
549 };
```

第 547 行的 `func` 函数就是 `tasklet` 要执行的处理函数, 用户实现具体的函数内容, 相当于中断处理函数。如果要使用 `tasklet`, 必须先定义一个 `tasklet_struct` 变量, 然后使用 `tasklet_init` 函数对其进行初始化, `tasklet_init` 函数原型如下:

```
void tasklet_init(struct tasklet_struct *t,
                 void (*func)(unsigned long),
                 unsigned long data);
```

函数参数和返回值含义如下:

t: 要初始化的 tasklet

func: tasklet 的处理函数。

data: 要传递给 func 函数的参数

返回值: 没有返回值。

也可以使用宏 DECLARE_TASKLET 来一次性完成 tasklet 的定义和初始化, DECLARE_TASKLET 定义在 include/linux/interrupt.h 文件中, 定义如下:

```
DECLARE_TASKLET(name, func, data)
```

其中 name 为要定义的 tasklet 名字, 其实就是 tasklet_struct 类型的变量名, func 就是 tasklet 的处理函数, data 是传递给 func 函数的参数。

在上半部, 也就是中断处理函数中调用 tasklet_schedule 函数就能使 tasklet 在合适的时间运行, tasklet_schedule 函数原型如下:

```
void tasklet_schedule(struct tasklet_struct *t)
```

函数参数和返回值含义如下:

t: 要调度的 tasklet, 也就是 DECLARE_TASKLET 宏里面的 name。

返回值: 没有返回值。

关于 tasklet 的参考使用示例如下所示:

示例代码 15.1.2.7 tasklet 使用示例

```
/* 定义 taselet          */
struct tasklet_struct testtasklet;

/* tasklet 处理函数      */
void testtasklet_func(unsigned long data)
{
    /* tasklet 具体处理内容 */
}

/* 中断处理函数 */
irqreturn_t test_handler(int irq, void *dev_id)
{
    .....
    /* 调度 tasklet          */
    tasklet_schedule(&testtasklet);
    .....
}

/* 驱动入口函数          */
static int __init xxxx_init(void)
{
    .....
    /* 初始化 tasklet          */
    tasklet_init(&testtasklet, testtasklet_func, data);
    /* 注册中断处理函数        */
    request_irq(xxx_irq, test_handler, 0, "xxx", &xxx_dev);
}
```

```
.....
```

```
}
```

2、工作队列

工作队列是另外一种下半部执行方式，工作队列在进程上下文执行，工作队列将要推后的工作交给一个内核线程去执行，因为工作队列工作在进程上下文，因此工作队列允许睡眠或重新调度。因此如果你要推后的工作可以睡眠那么就可以选择工作队列，否则的话就只能选择软中断或 tasklet。

Linux 内核使用 `work_struct` 结构体表示一个工作，内容如下(省略掉条件编译):

示例代码 15.1.2.8 `work_struct` 结构体

```
struct work_struct {
    atomic_long_t data;
    struct list_head entry;
    work_func_t func;          /* 工作队列处理函数 */
};
```

这些工作组织成工作队列，工作队列使用 `workqueue_struct` 结构体表示，内容如下(省略掉条件编译):

示例代码 15.1.2.9 `workqueue_struct` 结构体

```
struct workqueue_struct {
    struct list_head pwqs;
    struct list_head list;
    struct mutex mutex;
    int work_color;
    int flush_color;
    atomic_t nr_pwqs_to_flush;
    struct wq_flusher *first_flusher;
    struct list_head flusher_queue;
    struct list_head flusher_overflow;
    struct list_head maydays;
    struct worker *rescuer;
    int nr_drainers;
    int saved_max_active;
    struct workqueue_attrs *unbound_attrs;
    struct pool_workqueue *dfl_pwq;
    char name[WQ_NAME_LEN];
    struct rcu_head rcu;
    unsigned int flags ____cacheline_aligned;
    struct pool_workqueue __percpu *cpu_pwqs;
    struct pool_workqueue __rcu *numa_pwq_tbl[];
};
```

Linux 内核使用工作者线程(worker thread)来处理工作队列中的各个工作，Linux 内核使用 `worker` 结构体表示工作者线程，`worker` 结构体内容如下:

示例代码 15.1.2.10 `worker` 结构体

```
struct worker {
```

```

union {
    struct list_head    entry;
    struct hlist_node   hentry;
};

struct work_struct    *current_work;
work_func_t           current_func;
struct pool_workqueue *current_pwq;
struct list_head      scheduled;
struct task_struct    *task;
struct worker_pool    *pool;
struct list_head      node;
unsigned long         last_active;
unsigned int          flags;
int                   id;
int                   sleeping;
char                  desc[WORKER_DESC_LEN];
struct workqueue_struct *rescue_wq;
work_func_t           last_func;
};
    
```

从示例代码 15.1.2.10 可以看出, 每个 worker 都有一个工作队列, 工作者线程处理自己工作队列中的所有工作。在实际的驱动开发中, 我们只需要定义工作(work_struct)即可, 关于工作队列和工作者线程我们基本不用去管。简单创建工作很简单, 直接定义一个 work_struct 结构体变量即可, 然后使用 INIT_WORK 宏来初始化工作, INIT_WORK 宏定义如下:

```
#define INIT_WORK(_work, _func)
```

_work 表示要初始化的工作, _func 是工作对应的处理函数。

也可以使用 DECLARE_WORK 宏一次性完成工作的创建和初始化, 宏定义如下:

```
#define DECLARE_WORK(n, f)
```

n 表示定义的工作(work_struct), f 表示工作对应的处理函数。

和 tasklet 一样, 工作也是需要调度才能运行的, 工作的调度函数为 schedule_work, 函数原型如下所示:

```
bool schedule_work(struct work_struct *work)
```

函数参数和返回值含义如下:

work: 要调度的工作。

返回值: 0 成功, 其他值 失败。

关于工作队列的参考使用示例如下所示:

示例代码 15.1.2.11 工作队列使用示例

```

/* 定义工作(work)          */
struct work_struct testwork;

/* work 处理函数          */
void testwork_func_t(struct work_struct *work);
{
    
```

```

        /* work 具体处理内容      */
    }

    /* 中断处理函数      */
    irqreturn_t test_handler(int irq, void *dev_id)
    {
        .....
        /* 调度 work      */
        schedule_work(&testwork);
        .....
    }

    /* 驱动入口函数      */
    static int __init xxxx_init(void)
    {
        .....
        /* 初始化 work      */
        INIT_WORK(&testwork, testwork_func_t);
        /* 注册中断处理函数      */
        request_irq(xxx_irq, test_handler, 0, "xxx", &xxx_dev);
        .....
    }

```

15.1.3 设备树中断信息节点

1、GIC 中断控制器

GIC 全称为: Generic Interrupt Controller, 关于 GIC 的详细内容可以查看文档《ARM Generic Interrupt Controller(ARM GIC 控制器)V2.0》, 此文档已经放到了开发板光盘中, 路径为: [开发板光盘](#) → 4、[参考资料](#) → [ARM Generic Interrupt Controller\(ARM GIC 控制器\)V2.0.pdf](#)。

GIC 是 ARM 公司给 Cortex-A/R 内核提供的一个中断控制器, 类似 Cortex-M 内核中的 NVIC。目前 GIC 有 4 个版本:V1~V4, V1 是最老的版本, 已经被废弃了。V2~V4 目前正在大量的使用。GIC V2 是给 ARMv7-A 架构使用的, 比如 Cortex-A7、Cortex-A9、Cortex-A15 等, V3 和 V4 是给 ARMv8-A/R 架构使用的, 也就是 64 位芯片使用的。RK3568 是 Cortex-A55 内核, 因此我们主要讲解 GIC V2。GIC V2 最多支持 8 个核。ARM 会根据 GIC 版本的不同研发出不同的 IP 核, 那些半导体厂商直接购买对应的 IP 核即可, 比如 ARM 针对 GIC V2 就开发出了 GIC400 这个中断控制器 IP 核。当 GIC 接收到外部中断信号以后就会报给 ARM 内核, 但是 ARM 内核只提供了四个信号给 GIC 来汇报中断情况: VFIQ、VIRQ、FIQ 和 IRQ, 他们之间的关系如图 15.1.3.1 所示:

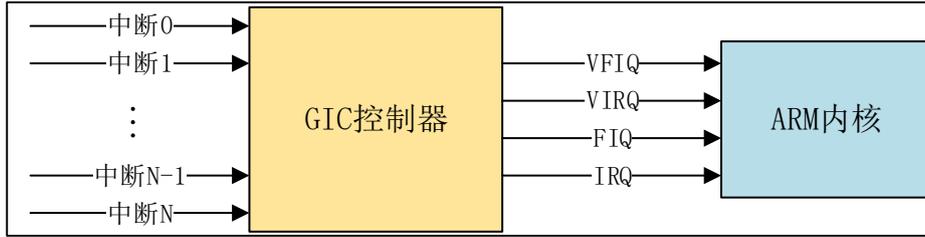


图 15.1.3.1 中断示意图

在图 15.1.3.1 中，GIC 接收众多的外部中断，然后对其进行处理，最终就只通过四个信号报给 ARM 内核，这四个信号的含义如下：

- VFIQ:**虚拟快速 FIQ。
- VIRQ:**虚拟快速 IRQ。
- FIQ:**快速中断 IRQ。
- IRQ:**外部中断 IRQ。

VFIQ 和 VIRQ 是针对虚拟化的，我们讨论不虚拟化，剩下的就是 FIQ 和 IRQ 了，本教程我们只使用 IRQ。所以相当于 GIC 最终向 ARM 内核就上报一个 IRQ 信号。那么 GIC 是如何完成这个工作的呢？GIC V2 的逻辑图如图 15.1.3.2 所示：

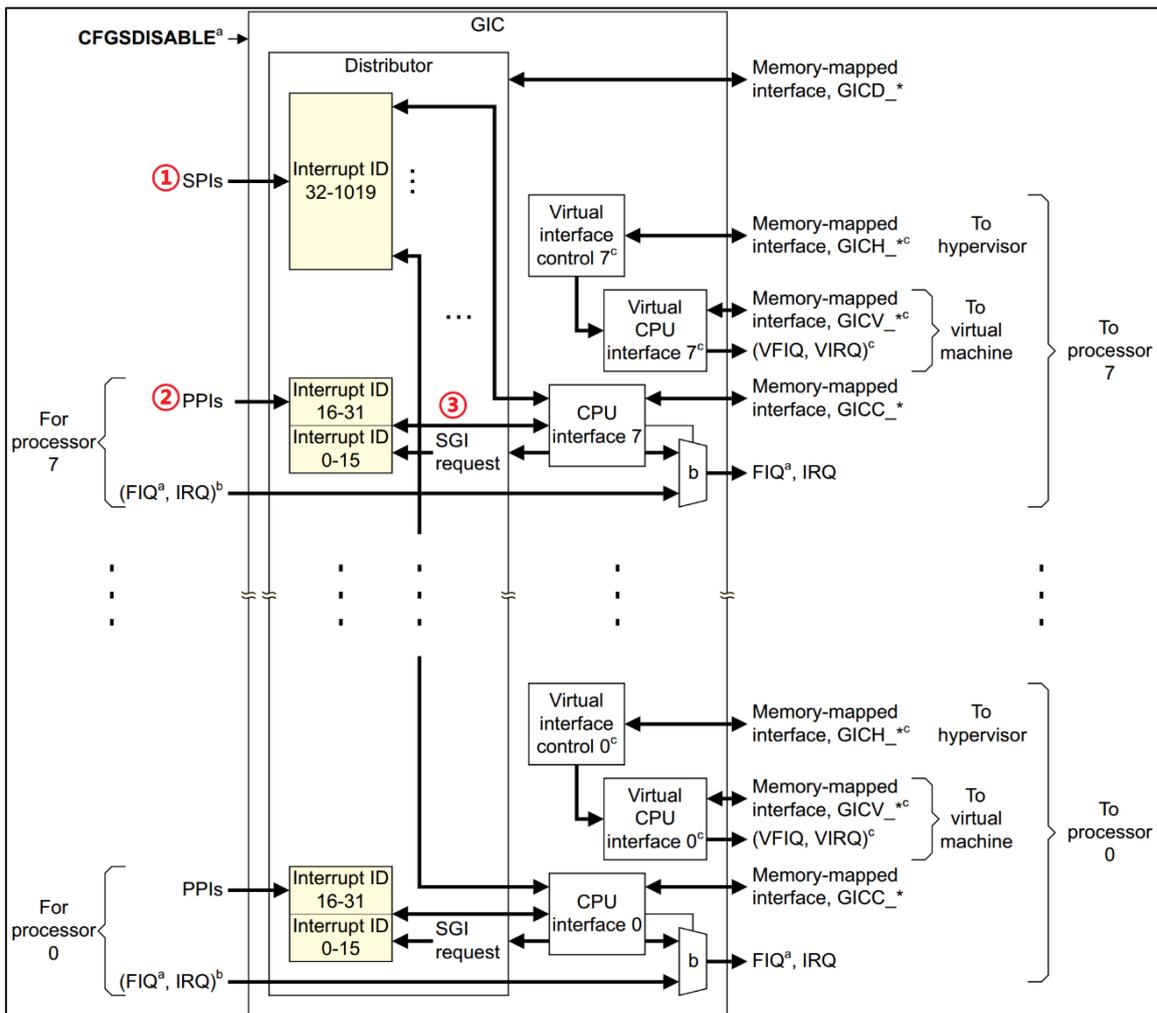


图 15.1.3.2 GIC V2 总体框图

图 15.1.3.2 中左侧部分就是中断源，中间部分就是 GIC 控制器，最右侧就是中断控制器向处理器内核发送中断信息。我们重点要看的肯定是中间的 GIC 部分，GIC 将众多的中断源分为分为三类：

①、SPI(Shared Peripheral Interrupt),共享中断，顾名思义，所有 Core 共享的中断，这个是最常见的，那些外部中断都属于 SPI 中断(注意！不是 SPI 总线那个中断)。比如 GPIO 中断、串口中断等等，这些中断所有的 Core 都可以处理，不限定特定 Core。

②、PPI(Private Peripheral Interrupt)，私有中断，我们说了 GIC 是支持多核的，每个核肯定有自己独有的中断。这些独有的中断肯定是要指定的核心处理，因此这些中断就叫做私有中断。

③、SGI(Software-generated Interrupt)，软件中断，由软件触发引起的中断，通过向寄存器 GICD_SGIR 写入数据来触发，系统会使用 SGI 中断来完成多核之间的通信。

2、中断 ID

中断源有很多，为了区分这些不同的中断源肯定要给他们分配一个唯一 ID，这些 ID 就是中断 ID。每一个 CPU 最多支持 1020 个中断 ID，中断 ID 号为 ID0~ID1019。这 1020 个 ID 包含了 PPI、SPI 和 SGI，那么这三类中断是如何分配这 1020 个中断 ID 的呢？这 1020 个 ID 分配如下：

ID0~ID15：这 16 个 ID 分配给 SGI。

ID16~ID31：这 16 个 ID 分配给 PPI。

ID32~ID1019：这 988 个 ID 分配给 SPI，像 GPIO 中断、串口中断等这些外部中断，至于具体到某个 ID 对应哪个中断那就由半导体厂商根据实际情况去定义了。比如 RK3568 的外设中断 ID 对应的中断源可以在《Rockchip RK3568 TRM Part1 V1.1 (RK3568 参考手册 1)》中找到详细的解释。找到“1.3 System Interrupt Connection”小节，中断 ID 如图 15.1.3.3 所示(由于表太大，这里只是截取其中一部分)：

Number	Source	Polarity
0-31 PPI		High level
32	audpwm	High level
33	can0	High level
34	can1	High level
35	can2	High level
36	crypto_ns	High level
37	Reserved	High level
38	csirx0_1	High level

图 15.1.3.3 RK3568 中断源

关于 GIC 就先讲到这里，我们接下来讲解一下 EXTI。

3、GIC 控制器节点

打开 rk3568.dtsi 文件，其中的 gic 节点就是 GIC 的中断控制器节点，节点内容如下所示：

示例代码 15.1.3.1 中断控制器 gic 节点

```

1  gic: interrupt-controller@fd400000 {
2      compatible = "arm,gic-v3";
3      #interrupt-cells = <3>;
4      #address-cells = <2>;
5      #size-cells = <2>;
6      ranges;
7      interrupt-controller;
8
9      reg = <0x0 0xfd400000 0 0x10000>, /* GICD */
10         <0x0 0xfd460000 0 0xc0000>; /* GICR */
11     interrupts = <GIC_PPI 9 IRQ_TYPE_LEVEL_HIGH>;
12     its: interrupt-controller@fd440000 {
13         compatible = "arm,gic-v3-its";
14         msi-controller;
15         #msi-cells = <1>;
16         reg = <0x0 0xfd440000 0x0 0x20000>;
17     };
18 };
    
```

第 2 行，compatible 属性值为“arm,gic-v3”在 Linux 内核源码中搜索“arm,gic-v3”即可找到 GIC 中断控制器驱动文件。

第 3 行，interrupt-controller 节点为空，表示当前节点是中断控制器。

第 4 行，#interrupt-cells 和 #address-cells、#size-cells 一样。表示此中断控制器下设备的 cells 大小，对于设备而言，会使用 interrupts 属性描述中断信息，#interrupt-cells 描述了 interrupts 属性的 cells 大小，也就是一条信息有几个 cells。每个 cells 都是 32 位整形值，对于 ARM 处理的 GIC 来说，一共有 3 个 cells，这三个 cells 的含义如下：

第一个 cells：中断类型，0 表示 SPI 中断，1 表示 PPI 中断。

第二个 cells：中断号，对于 SPI 中断来说中断号的范围为 32~1019（具体取决于半导体厂商实际使用了多少个中断号），对于 PPI 中断来说中断号的范围为 16~31，但是该 cell 描述的中断号是从 0 开始。

第三个 cells：标志，bit[3:0]表示中断触发类型，为 1 的时候表示上升沿触发，为 2 的时候表示下降沿触发，为 4 的时候表示高电平触发，为 8 的时候表示低电平触发。bit[15:8]为 PPI 中断的 CPU 掩码。

我们来看一下 RK3568 的 SPI0 是如何在设备树节点中描述中断信息的，首先是查阅《Rockchip RK3568 TRM Part1 V1.1 (RK3568 参考手册 1)》第“1.3 System Interrupt Connection”小节中的表 1-3。找到 SPI0 对应的中断号，如图 15.1.3.4 所示：

135	spi0	High level
136	spi1	High level
137	spi2	High level

图 15.1.3.4 SPI0 中断

从图 15.1.3.6 可以看出，SPI0 的中断号为 135，注意这里是加上了前面 32 个 PPI 中断号，如果不算前面 32 个 PPI 中断号的话就是就是 135-32=103。

打开 rk3568.dtsi，找到 SPI0 节点内容，如下所示：

示例代码 15.1.3.2 SPI0 节点

```

1 spi0: spi@fe610000 {
2     compatible = "rockchip,rk3066-spi";
3     reg = <0x0 0xfe610000 0x0 0x1000>;
4     interrupts = <GIC_SPI 103 IRQ_TYPE_LEVEL_HIGH>;
5     #address-cells = <1>;
6     #size-cells = <0>;
7     clocks = <&cru CLK_SPI0>, <&cru PCLK_SPI0>;
8     clock-names = "spiclk", "apb_pclk";
9     dmas = <&dmac0 20>, <&dmac0 21>;
10    dma-names = "tx", "rx";
11    pinctrl-names = "default", "high_speed";
12    pinctrl-0 = <&spi0m0_cs0 &spi0m0_cs1 &spi0m0_pins>;
13    pinctrl-1 = <&spi0m0_cs0 &spi0m0_cs1 &spi0m0_pins_hs>;
14    status = "disabled";
15};

```

第 4 行，interrupts 描述中断源的信息，第一个表示中断类型，为 GIC_SPI，也就是共享中断。第二个表示中断号为 103，来源就是图 15.1.3.4 中的 135-32=103。第三个表示中断触发类型是高电平触发。

我们来看一个具体的应用，打开 rk3568-evb.dtsi 文件，找到如下所示内容：

示例代码 15.1.3.6 hdmi 节点信息

```

1095 &i2c0 {
1096     status = "okay";
1097
1098     .....
1115
1116     rk809: pmic@20 {
1117         compatible = "rockchip,rk809";
1118         reg = <0x20>;
1119         interrupt-parent = <&gpio0>;
1120         interrupts = <3 IRQ_TYPE_LEVEL_LOW>;
1121

```

```

1122         pinctrl-names = "default", "pmic-sleep",
1123         "pmic-power-off", "pmic-reset";
.....
1378     };
1379 };
    
```

RK809 是正点原子 ATK-DLRK3568 开发板上核心板的 PMIC 芯片，上述代码就是 RK809 的节点信息，RK809 芯片有一个中断，此引脚链接到了 RK3568 的 GPIO0_A3 上，此中断是电平触发。

第 1119 行，`interrupt-parent` 属性设置中断控制器，因为 GPIO0_A3 数据 GPIO0 组，所以这里设置中断控制器为 GPIO0。

第 1120 行，`interrupts` 设置中断信息，3 表示 GPIO0_A3 属于 GPIO0 组的第 4 个 IO，前 3 个为 A0~A2。IRQ_TYPE_LEVEL_LOW 表示下降沿触发。

结合第 1119,1120 这两行，目的就是设置 GPIO0_A3 为下低电平触发。可以看出使用起来是非常的简单，在我们实际编写代码的时候，只需要通过 `interrupt-parent` 和 `interrupts` 这两个属性即可设置某个 GPIO 的中断功能。

简单总结一下与中断有关的设备树属性信息：

- ①、`#interrupt-cells`，指定中断源的信息 `cells` 个数。
- ②、`interrupt-controller`，表示当前节点为中断控制器。
- ③、`interrupts`，指定中断号，触发方式等。
- ④、`interrupt-parent`，指定父中断，也就是中断控制器。

15.1.4 获取中断号

编写驱动的时候需要用到中断号，我们用到的中断号，中断信息已经写到了设备树里面，因此可以通过 `irq_of_parse_and_map` 函数从 `interrupts` 属性中提取到对应的设备号，函数原型如下：

```
unsigned int irq_of_parse_and_map(struct device_node *dev, int index)
```

函数参数和返回值含义如下：

dev: 设备节点。

index: 索引号，`interrupts` 属性可能包含多条中断信息，通过 `index` 指定要获取的信息。

返回值：中断号。

如果使用 GPIO 的话，可以使用 `gpio_to_irq` 函数来获取 `gpio` 对应的中断号，函数原型如下：

```
int gpio_to_irq(unsigned int gpio)
```

函数参数和返回值含义如下：

gpio: 要获取的 GPIO 编号。

返回值：GPIO 对应的中断号。

15.2 硬件原理图分析

本章实验硬件原理图参考 13.2 小节即可。本章我们依旧使用《第十三章 Linux 按键输入实验》里面的 GPIO3_C5 引脚来模拟按键。

15.3 实验程序编写

本实验对应的例程路径为：[开发板光盘](#)→01、[程序源码](#)→Linux 驱动例程→12_irq。

本章实验我们驱动正点原子的 ATK-DLRK3568 开发板上的 GPIO3_C5 引脚来模拟按键，不过我们采用中断的方式，并且采用定时器来实现按键消抖，应用程序读取按键值并且通过终端打印出来。通过本章我们可以学习到 Linux 内核中断的使用方法，以及对 Linux 内核定时器的回顾。

15.3.1 修改设备树文件

本章实验我们在第十三章的示例代码 13.3.1.2 创建的 key 节点基础上补充，添加 GPIO3_C5 这个引脚相关的中断属性，添加完成以后的“key”节点内容如下所示：

示例代码 15.3.1.1 key 节点信息

```

1 key {
2     compatible = "alientek,key";
3     pinctrl-names = "alientek,key";
4     pinctrl-0 = <&key_gpio>;
5     key-gpio = <&gpio3 RK_PC5 GPIO_ACTIVE_HIGH>;
6     interrupt-parent = <&gpio3>;
7     interrupts = <21 IRQ_TYPE_EDGE_BOTH>;
8     status = "okay";
9 };
    
```

第 6 行，设置 interrupt-parent 属性值为“gpio3”。

第 7 行，设置 interrupts 属性，也就是设置中断源，第一个 cells 的 21 表示 GPIO3 组的 21 号 IO，也就是 PC5。IRQ_TYPE_EDGE_BOTH 定义在文件 include/linux/irq.h 中，定义如下：

示例代码 15.3.1.2 中断状态

```

74 enum {
75     IRQ_TYPE_NONE                = 0x00000000,
76     IRQ_TYPE_EDGE_RISING        = 0x00000001,
77     IRQ_TYPE_EDGE_FALLING       = 0x00000002,
78     IRQ_TYPE_EDGE_BOTH          = (IRQ_TYPE_EDGE_FALLING |
79                                   IRQ_TYPE_EDGE_RISING),
79     IRQ_TYPE_LEVEL_HIGH         = 0x00000004,
80     IRQ_TYPE_LEVEL_LOW          = 0x00000008,
81     IRQ_TYPE_LEVEL_MASK         = (IRQ_TYPE_LEVEL_LOW |
82                                   IRQ_TYPE_LEVEL_HIGH),
83     .....
84     .....
85     .....
86     .....
87     .....
88     .....
89 };
    
```

从示例代码 15.3.1.2 中可以看出，IRQ_TYPE_EDGE_BOTH 表示上升沿和下降沿同时有效，相当于按下和释放都会触发中断。

设备树修改完成以后，重新编译内核，并且烧写到开发板中。

15.3.2 按键中断驱动程序编写

新建名为“12_irq”的文件夹，然后在 12_irq 文件夹里面创建 vscode 工程，工作区命名为“keyirq”。工程创建好以后新建 keyirq.c 文件，在 keyirq.c 里面输入如下内容：

示例代码 15.3.2.1 keyirq.c 文件代码

```

1  /*****
2  Copyright © ALIENTEK Co., Ltd. 1998-2029. All rights reserved.
3  文件名   : key.c
4  作者     : 正点原子 Linux 团队
5  版本     : V1.0
6  描述     : Linux 中断驱动实验
7  其他     : 无
8  论坛     : www.openedv.com
9  日志     : 初版 v1.0 2022/12/28 正点原子 Linux 团队创建
10 *****/
11 #include <linux/types.h>
12 #include <linux/kernel.h>
13 #include <linux/delay.h>
14 #include <linux/ide.h>
15 #include <linux/init.h>
16 #include <linux/module.h>
17 #include <linux/errno.h>
18 #include <linux/gpio.h>
19 #include <linux/cdev.h>
20 #include <linux/device.h>
21 #include <linux/of.h>
22 #include <linux/of_address.h>
23 #include <linux/of_gpio.h>
24 #include <linux/semaphore.h>
25 #include <linux/of_irq.h>
26 #include <linux/irq.h>
27 // #include <asm/mach/map.h>
28 #include <asm/uaccess.h>
29 #include <asm/io.h>
30
31 #define KEY_CNT      1          /* 设备号个数 */
32 #define KEY_NAME     "key"     /* 名字 */
33
34 /* 定义按键状态 */
35 enum key_status {
36     KEY_PRESS = 0,           /* 按键按下 */
37     KEY_RELEASE,           /* 按键松开 */
38     KEY_KEEP,              /* 按键状态保持 */

```

```

39 };
40
41 /* key 设备结构体 */
42 struct key_dev{
43     dev_t devid;           /* 设备号          */
44     struct cdev cdev;     /* cdev            */
45     struct class *class;  /* 类              */
46     struct device *device; /* 设备            */
47     struct device_node *nd; /* 设备节点        */
48     int key_gpio;         /* key 所使用的 GPIO 编号 */
49     struct timer_list timer; /* 按键值          */
50     int irq_num;          /* 中断号          */
51     spinlock_t spinlock; /* 自旋锁          */
52 };
53
54 static struct key_dev key; /* 按键设备        */
55 static int status = KEY_KEEP; /* 按键状态        */
56
57 static irqreturn_t key_interrupt(int irq, void *dev_id)
58 {
59     /* 按键防抖处理, 开启定时器延时 15ms */
60     mod_timer(&key.timer, jiffies + msecs_to_jiffies(15));
61     return IRQ_HANDLED;
62 }
63
64 /*
65  * @description   : 初始化按键 IO, open 函数打开驱动的时候初始化按键所使
66  *                  用的 GPIO 引脚。
67  * @param         : 无
68  * @return        : 无
69  */
70 static int key_parse_dt(void)
71 {
72     int ret;
73     const char *str;
74
75     /* 设置 LED 所使用的 GPIO */
76     /* 1、获取设备节点: key */
77     key.nd = of_find_node_by_path("/key");
78     if(key.nd == NULL) {
79         printk("key node not find!\r\n");
80         return -EINVAL;
81     }

```

```

82
83     /* 2.读取 status 属性 */
84     ret = of_property_read_string(key.nd, "status", &str);
85     if(ret < 0)
86         return -EINVAL;
87
88     if (strcmp(str, "okay"))
89         return -EINVAL;
90
91     /* 3、获取 compatible 属性值并进行匹配 */
92     ret = of_property_read_string(key.nd, "compatible", &str);
93     if(ret < 0) {
94         printk("key: Failed to get compatible property\n");
95         return -EINVAL;
96     }
97
98     if (strcmp(str, "alientek,key")) {
99         printk("key: Compatible match failed\n");
100        return -EINVAL;
101    }
102
103    /* 4、 获取设备树中的 gpio 属性, 得到 KEY0 所使用的 KEY 编号 */
104    key.key_gpio = of_get_named_gpio(key.nd, "key-gpio", 0);
105    if(key.key_gpio < 0) {
106        printk("can't get key-gpio");
107        return -EINVAL;
108    }
109
110    /* 5、获取 GPIO 对应的中断号 */
111    key.irq_num = irq_of_parse_and_map(key.nd, 0);
112    if(!key.irq_num){
113        return -EINVAL;
114    }
115
116    printk("key-gpio num = %d\r\n", key.key_gpio);
117    return 0;
118 }
119
120 static int key_gpio_init(void)
121 {
122     int ret;
123     unsigned long irq_flags;
124

```

```

125     ret = gpio_request(key.key_gpio, "KEY0");
126     if (ret) {
127         printk(KERN_ERR "key: Failed to request key-gpio\n");
128         return ret;
129     }
130
131     /* 将 GPIO 设置为输入模式 */
132     gpio_direction_input(key.key_gpio);
133
134     /* 获取设备树中指定的中断触发类型 */
135     irq_flags = irq_get_trigger_type(key.irq_num);
136     if (irq_flags == IRQF_TRIGGER_NONE)
137         irq_flags = IRQF_TRIGGER_FALLING | IRQF_TRIGGER_RISING;
138
139     /* 申请中断 */
140     ret = request_irq(key.irq_num, key_interrupt, irq_flags,
141                      "Key0_IRQ", NULL);
142     if (ret) {
143         gpio_free(key.key_gpio);
144         return ret;
145     }
146     return 0;
147 }
148
149 static void key_timer_function(struct timer_list *arg)
150 {
151     static int last_val = 0;
152     unsigned long flags;
153     int current_val;
154
155     /* 自旋锁上锁 */
156     spin_lock_irqsave(&key.spinlock, flags);
157
158     /* 读取按键值并判断按键当前状态 */
159     current_val = gpio_get_value(key.key_gpio);
160     if (1 == current_val && !last_val)    /* 按下    */
161         status = KEY_PRESS;
162     else if (0 == current_val && last_val)
163         status = KEY_RELEASE;            /* 松开    */
164     else
165         status = KEY_KEEP;                /* 状态保持 */
166

```

```

167     last_val = current_val;
168
169     /* 自旋锁解锁 */
170     spin_unlock_irqrestore(&key.spinlock, flags);
171 }
172
173 /*
174 * @description   : 打开设备
175 * @param - inode: 传递给驱动的 inode
176 * @param - filp  : 设备文件, file 结构体有个叫做 private_data 的成员变量
177 *                  一般在 open 的时候将 private_data 指向设备结构体。
178 * @return        : 0 成功;其他 失败
179 */
180 static int key_open(struct inode *inode, struct file *filp)
181 {
182     return 0;
183 }
184
185 /*
186 * @description   : 从设备读取数据
187 * @param - filp  : 要打开的设备文件 (文件描述符)
188 * @param - buf   : 返回给用户空间的数据缓冲区
189 * @param - cnt   : 要读取的数据长度
190 * @param - offt  : 相对于文件首地址的偏移
191 * @return        : 读取的字节数, 如果为负值, 表示读取失败
192 */
193 static ssize_t key_read(struct file *filp, char __user *buf,
194                        size_t cnt, loff_t *offt)
195 {
196     unsigned long flags;
197     int ret;
198
199     /* 自旋锁上锁 */
200     spin_lock_irqsave(&key.spinlock, flags);
201
202     /* 将按键状态信息发送给应用程序 */
203     ret = copy_to_user(buf, &status, sizeof(int));
204
205     /* 状态重置 */
206     status = KEY_KEEP;
207
208     /* 自旋锁解锁 */
209     spin_unlock_irqrestore(&key.spinlock, flags);

```

```

210
211     return ret;
212 }
213
214 /*
215 * @description   : 向设备写数据
216 * @param - filp : 设备文件, 表示打开的文件描述符
217 * @param - buf  : 要写给设备写入的数据
218 * @param - cnt  : 要写入的数据长度
219 * @param - offt : 相对于文件首地址的偏移
220 * @return        : 写入的字节数, 如果为负值, 表示写入失败
221 */
222 static ssize_t key_write(struct file *filp, const char __user *buf,
                          size_t cnt, loff_t *offt)
223 {
224     return 0;
225 }
226
227 /*
228 * @description   : 关闭/释放设备
229 * @param - filp : 要关闭的设备文件(文件描述符)
230 * @return        : 0 成功;其他 失败
231 */
232 static int key_release(struct inode *inode, struct file *filp)
233 {
234     return 0;
235 }
236
237 /* 设备操作函数 */
238 static struct file_operations key_fops = {
239     .owner = THIS_MODULE,
240     .open = key_open,
241     .read = key_read,
242     .write = key_write,
243     .release = key_release,
244 };
245
246 /*
247 * @description   : 驱动入口函数
248 * @param        : 无
249 * @return        : 无
250 */
251 static int __init mykey_init(void)

```

```

252 {
253     int ret;
254
255     /* 初始化自旋锁 */
256     spin_lock_init(&key.spinlock);
257
258     /* 1、初始化 timer, 设置定时器处理函数, 还未设置周期, 所以不会激活定时器 */
259     timer_setup(&key.timer, key_timer_function, 0);
260
261     /* 2、设备树解析 */
262     ret = key_parse_dt();
263     if(ret)
264         return ret;
265
266     /* 3、GPIO 中断初始化 */
267     ret = key_gpio_init();
268     if(ret)
269         return ret;
270
271     /* 注册字符设备驱动 */
272     /* 1、创建设备号 */
273     ret = alloc_chrdev_region(&key.devid, 0, KEY_CNT, KEY_NAME);
274     if(ret < 0) {
275         pr_err("%s Couldn't alloc_chrdev_region, ret=%d\r\n",
276                KEY_NAME, ret);
277         goto free_gpio;
278     }
279
280     /* 2、初始化 cdev */
281     key.cdev.owner = THIS_MODULE;
282     cdev_init(&key.cdev, &key_fops);
283
284     /* 3、添加一个 cdev */
285     ret = cdev_add(&key.cdev, key.devid, KEY_CNT);
286     if(ret < 0)
287         goto del_unregister;
288
289     /* 4、创建类 */
290     key.class = class_create(THIS_MODULE, KEY_NAME);
291     if (IS_ERR(key.class)) {
292         goto del_cdev;
293     }

```

```

294     /* 5、创建设备 */
295     key.device = device_create(key.class, NULL, key.devid, NULL,
                                KEY_NAME);
296     if (IS_ERR(key.device)) {
297         goto destroy_class;
298     }
299
300     return 0;
301
302 destroy_class:
303     class_destroy(key.class);
304 del_cdev:
305     cdev_del(&key.cdev);
306 del_unregister:
307     unregister_chrdev_region(key.devid, KEY_CNT);
308 free_gpio:
309     free_irq(key.irq_num, NULL);
310     gpio_free(key.key_gpio);
311     return -EIO;
312 }
313
314 /*
315  * @description   : 驱动出口函数
316  * @param         : 无
317  * @return        : 无
318  */
319 static void __exit mykey_exit(void)
320 {
321     /* 注销字符设备驱动 */
322     cdev_del(&key.cdev);          /* 删除 cdev */
323     unregister_chrdev_region(key.devid, KEY_CNT); /* 注销设备号 */
324     del_timer_sync(&key.timer);  /* 删除 timer */
325     device_destroy(key.class, key.devid); /* 注销设备 */
326     class_destroy(key.class);     /* 注销类 */
327     free_irq(key.irq_num, NULL);  /* 释放中断 */
328     gpio_free(key.key_gpio);     /* 释放 IO */
329 }
330
331 module_init(mykey_init);
332 module_exit(mykey_exit);
333 MODULE_LICENSE("GPL");
334 MODULE_AUTHOR("ALIENTEK");
335 MODULE_INFO(intree, "Y");
    
```

第 35~39 行, 定义了一个枚举类型, 包含 3 个常量 KEY_PRESS、KEY_RELEASE、KEY_KEEP, 分别用来表示按键的 3 种不同的状态, 即按键按下、按键松开以及按键状态保持。

第 42~52 行, 结构体 key_dev 为按键设备所对应的结构体, key_gpio 为按键 GPIO 编号, irq_num 为按键 IO 对应的中断号; 除此之外, 结构体当中还定义了一个定时器用于实现按键的去抖操作, 还定义了一个自旋锁用于实现对关键代码的保护操作。

第 54 行, 定义一个按键设备 key。

第 55 行, 定义一个 int 类型的静态全局变量 status 用来表示按键的状态。

第 57~62 行, key_interrupt 函数是中断处理函数, 参数 dev_id 是一个 void 类型的指针, 本驱动程序并没使用到这个参数; 这个中断处理函数很简单直接开启定时器, 延时 15 毫秒, 用于实现按键的软件防抖。

第 70~118 行, key_parse_dt 函数中主要是对设备树中的属性进行了解析, 获取设备树中的 key 节点, 通过 of_get_named_gpio 函数得到按键的 GPIO 编号, 通过 irq_of_parse_and_map 函数获取按键的中断号, irq_of_parse_and_map 函数会解析 key 节点中的 interrupt-parent 和 interrupts 属性然后得到一个中断号, 后面就可以使用这个中断号去申请以及释放中断了。

第 120~147 行, key_gpio_init 函数中主要对 GPIO 以及中断进行了相关的初始化。使用 gpio_request 函数申请 GPIO 使用权, 通过 gpio_direction_input 将 GPIO 设置为输入模式; irq_get_trigger_type 函数可以获取到 key 节点中定义的中断触发类型, 最后使用 request_irq 申请中断, 并设置 key_interrupt 函数作为我们的按键中断处理函数, 当按键中断发生之后便会跳转到该函数执行; request_irq 函数会默认使能中断, 所以不需要 enable_irq 来使能中断, 当然, 我们也可以在申请成功之后先使用 disable_irq 函数禁用中断, 等所有工作完成之后再使能中断, 这样会比较安全, 建议大家这样使用。

第 149~171 行, key_timer_function 函数为定时器定时处理函数, 它的参数 arg 在本驱动程序中我们并没有使用到; 该函数中定义了一个静态局部变量 last_val 用来保存按键上一次读取到的值, 变量 current_val 用来存放当前按键读取到的值; 第 159~165 行, 通过读取到的按键值以及上一次读取到的值来判断按键当前所属的状态, 如果本次读取的值为 1, 而上一次读取的值 0, 则表示按键按下; 如果本次读取的值为 0, 而上一次读取的值 1, 则表示按键松开; 如果本次读取的值为 1, 而上一次读取的值也是 1, 则表示按键一直被按着; 如果本次读取的值 0, 而上一次读取的值也是 0, 则表示没有触碰按键。第 167 行, 当状态判断完成之后, 会将 current_val 的值赋值给 last_val。本函数中也使用自旋锁对全局变量 status 进行加锁保护!

第 193~212 行, key_read 函数, 对应应用程序的 read 函数。此函数向应用程序返回按键状态信息数据; 这个函数其实很简单, 使用 copy_to_user 函数直接将 status 数据发送给应用程序, status 变量保存了按键当前的状态, 发送完成之后再按键状态重置即可! 需要注意的是, 该函数中使用了自旋锁进行保护。

第 238~244 行, 按键设备的 file_operations 结构体。

第 251~312 行, mykey_init 是驱动入口函数, 第 256 行调用 spin_lock_init 初始化自旋锁变量, 259 行对定时器进行初始化并将 key_timer_function 函数绑定为定时器定时处理函数, 当定时时间到了之后便会跳转到该函数执行。

第 319~329 行, mykey_exit 驱动出口函数, 第 324 行调用 del_timer_sync 函数删除定时器, 代码中已经注释得非常详细了, 这里便不再多说!

15.3.2 编写测试 APP

测试 APP 要实现的内容很简单, 通过不断的读取/dev/key 设备文件来获取按键值来判断当前按键的状态, 从按键驱动上传到应用程序的数据可以有 3 个值, 分别为 0、1、2; 0 表示按键

按下时的这个状态，1 表示按键松开时对应的状态，而 2 表示按键一直被按住或者松开；搞懂数据代表的意义之后，我们开始编写测试程序，在 12_irq 目录下新建名为 keyirqApp.c 的文件，然后输入如下所示内容：

示例代码 15.3.3.1 keyrqApp.c 文件代码

```

1  /*****
2  Copyright © ALIENTEK Co., Ltd. 1998-2029. All rights reserved.
3  文件名      : keyApp.c
4  作者        : 正点原子 Linux 团队
5  版本        : V1.0
6  描述        : Linux 中断驱动实验
7  其他        : 无
8  使用方法    : ./keyirqApp /dev/key
9  论坛        : www.openedv.com
10  日志        : 初版 V1.0 2022/12/29 正点原子 Linux 团队创建
11  *****/
12
13 #include <stdio.h>
14 #include <unistd.h>
15 #include <sys/types.h>
16 #include <sys/stat.h>
17 #include <fcntl.h>
18 #include <stdlib.h>
19 #include <string.h>
20
21 /*
22  * @description : main 主程序
23  * @param - argc : argv 数组元素个数
24  * @param - argv : 具体参数
25  * @return      : 0 成功;其他 失败
26  */
27 int main(int argc, char *argv[])
28 {
29     int fd, ret;
30     int key_val;
31
32     /* 判断传参个数是否正确 */
33     if(2 != argc) {
34         printf("Usage:\n"
35             "\t./keyApp /dev/key\n"
36             );
37         return -1;
38     }
39

```

```

40     /* 打开设备 */
41     fd = open(argv[1], O_RDONLY);
42     if(0 > fd) {
43         printf("ERROR: %s file open failed!\n", argv[1]);
44         return -1;
45     }
46
47     /* 循环读取按键数据 */
48     for ( ; ; ) {
49
50         read(fd, &key_val, sizeof(int));
51         if (0 == key_val)
52             printf("Key Press\n");
53         else if (1 == key_val)
54             printf("Key Release\n");
55     }
56
57     /* 关闭设备 */
58     close(fd);
59     return 0;
60 }
    
```

第 48~55 行使用 for 循环不断的读取按键值，如果读取到的值是 0 则打印“Key Press”字符串，而过读取到的值是 1 则打印“Key Release”字符串。

15.4 运行测试

15.4.1 编译驱动程序和测试 APP

1、编译驱动程序

编写 Makefile 文件，本章实验的 Makefile 文件和第五章实验基本一样，只是将 obj-m 变量的值改为 keyirq.o，Makefile 内容如下所示：

示例代码 15.4.1.1 Makefile 文件

```

1  KERNELDIR := /home/alientek/rk3568_linux_sdk/kernel
.....
4  obj-m := keyirq.o
.....
11 clean:
12 $(MAKE) -C $(KERNELDIR) M=$(CURRENT_PATH) clean
    
```

第 4 行，设置 obj-m 变量的值为 keyirq.o。

输入如下命令编译出驱动模块文件：

```
make ARCH=arm64 //ARCH=arm64 必须指定，否则编译会失败
```

编译成功以后就会生成一个名为“keyirq.ko”的驱动模块文件。

2、编译测试 APP

输入如下命令编译测试 keyirqApp.c 这个测试程序:

```
/opt/atk-dlrk356x-toolchain/bin/aarch64-buildroot-linux-gnu-gcc keyirqApp.c -o keyirqApp
编译成功以后就会生成 keyirqApp 这个应用程序。
```

15.4.2 运行测试

在 Ubuntu 中将上一小节编译出来的 keyirq.ko 和 keyirqApp 这两个文件通过 adb 命令发送到开发板的/lib/modules/4.19.232 目录下, 命令如下:

```
adb push keyirq.ko keyirqApp /lib/modules/4.19.232
```

发送成功以后进入到开发板目录 lib/modules/4.19.232 中, 输入如下命令加载 keyirq.ko 驱动模块:

```
depmod //第一次加载驱动的时候需要运行此命令
modprobe keyirq //加载驱动
```

驱动加载成功以后可以通过查看 /proc/interrupts 文件来检查一下对应的中断有没有被注册上, 输入如下命令:

```
cat /proc/interrupts
```

结果如图 15.4.2.1 所示:

82:	0	0	0	0	gpio0	3	Level	rk817
83:	0	0	0	0	rk817	0	Edge	rk805_pwrkey_fall
84:	0	0	0	0	rk817	1	Edge	rk805_pwrkey_rise
88:	0	0	0	0	rk817	5	Edge	RTC alarm
107:	0	0	0	0	gpio0	27	Level	rtc-pcf8563
108:	0	0	0	0	gpio0	15	Edge	bt_default_wake_host_irq
109:	0	0	0	0	gpio1	4	Edge	Headphone detection
110:	0	0	0	0	gpio3	21	Edge	Key0 IRQ
IPI0:	2297	3540	4792	5576				Rescheduling interrupts
IPI1:	1098	1408	1377	1583				Function call interrupts
IPI2:	0	0	0	0				CPU stop interrupts
IPI3:	0	0	0	0				CPU stop (for crash dump) interrupts

图 15.4.2.1 proc/interrupts 文件内容

从图 15.4.2.1 可以看出 keyirq.c 驱动文件里面的 KEY0 中断已经存在了, 触发方式为跳边沿(Edge)。

接下来使用如下命令来测试中断:

```
./keyirqApp /dev/key
```

使用杜邦线将图 13.2.1 中 GPIO3_C5 这个 IO 接到开发板的 3.3V 电压上, 模拟按键被按下, 终端就会输出按键值, 如图 15.4.2.2 所示:

```
root@ATK-DLRK356X:/lib/modules/4.19.232# ./keyirqApp /dev/key
Key Press
Key Release
Key Press
Key Release
Key Press
Key Release
Key Press
Key Release
```

图 15.4.2.2 读取到的按键值

从图 15.4.2.2 可以看出, 按键值获取成功, 并且不会有按键抖动导致的误判发生, 说明按键消抖工作正常。如果要卸载驱动的话输入如下命令即可:

```
rmmmod keyirq.ko
```

第十六章 Linux 阻塞和非阻塞 IO 实验

阻塞和非阻塞 IO 是 Linux 驱动开发里面很常见的两种设备访问模式，在编写驱动的时候一定要考虑到阻塞和非阻塞。本章我们就来学习一下阻塞和非阻塞 IO，以及如何在驱动程序中处理阻塞与非阻塞，如何在驱动程序使用等待队列和 poll 机制。

16.1 阻塞和非阻塞 IO

16.1.1 阻塞和非阻塞简介

这里的“IO”并不是我们学习单片机的时候所说的“GPIO”(也就是引脚)。这里的 IO 指的是 Input/Output, 也就是输入/输出, 是应用程序对驱动设备的输入/输出操作。当应用程序对设备驱动进行操作的时候, 如果不能获取到设备资源, 那么阻塞式 IO 就会将应用程序对应的线程挂起, 直到设备资源可以获取为止。对于非阻塞 IO, 应用程序对应的线程不会挂起, 它要么一直轮询等待, 直到设备资源可以使用, 要么就直接放弃。阻塞式 IO 如图 16.1.1.1 所示:

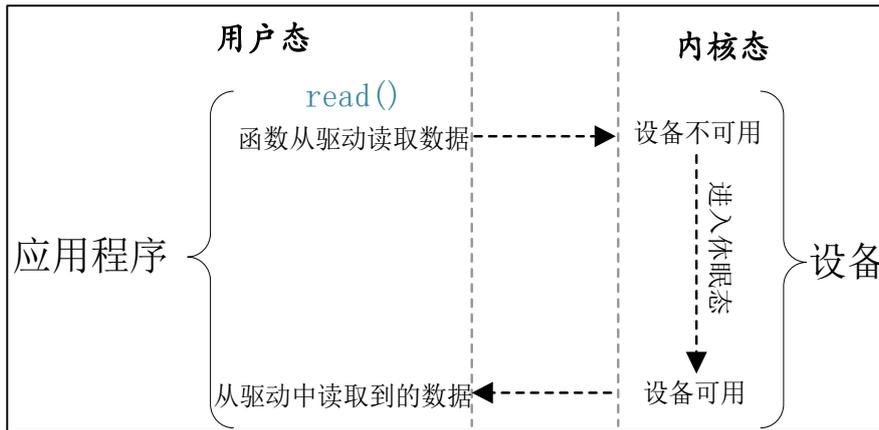


图 16.1.1.1 阻塞 IO 访问示意图。

图 16.1.1.1 中应用程序调用 read 函数从设备中读取数据, 当设备不可用或数据未准备好的时候就会进入到休眠态。等设备可用的时候就会从休眠态唤醒, 然后从设备中读取数据返回给应用程序。非阻塞 IO 如图 16.1.2 所示:

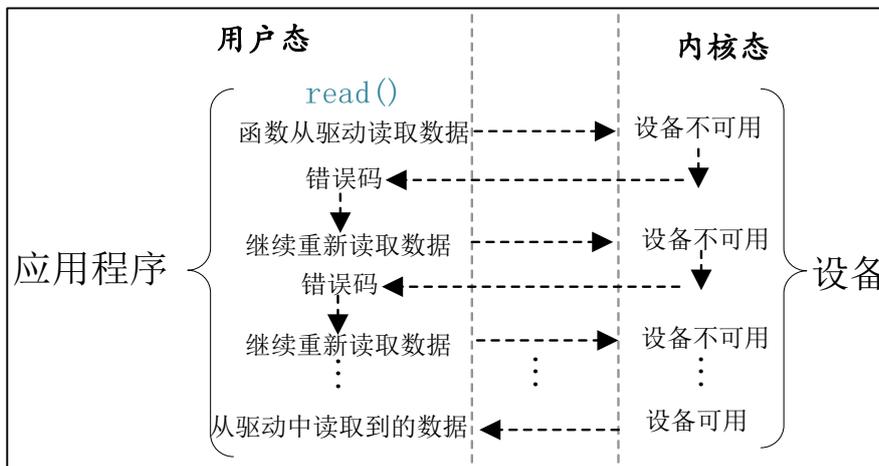


图 16.1.1.2 非阻塞 IO 访问示意图

从图 16.1.1.2 可以看出, 应用程序使用非阻塞访问方式从设备读取数据, 当设备不可用或数据未准备好的时候会立即向内核返回一个错误码, 表示数据读取失败。应用程序会再次重新读取数据, 这样一直往复循环, 直到数据读取成功。

应用程序可以使用如下所示示例代码来实现阻塞访问:

示例代码 16.1.1.1 应用程序阻塞读取数据

```
1 int fd;
2 int data = 0;
```

```

3
4 fd = open("/dev/xxx_dev", O_RDWR);          /* 阻塞方式打开 */
5 ret = read(fd, &data, sizeof(data));        /* 读取数据 */
    
```

从示例代码 16.1.1.1 可以看出, 对于设备驱动文件的默认读取方式就是阻塞式的, 所以我们前面所有的例程测试 APP 都是采用阻塞 IO。

如果应用程序要采用非阻塞的方式来访问驱动设备文件, 可以使用如下所示代码:

示例代码 16.1.1.2 应用程序非阻塞读取数据

```

1 int fd;
2 int data = 0;
3
4 fd = open("/dev/xxx_dev", O_RDWR | O_NONBLOCK); /* 非阻塞方式打开 */
5 ret = read(fd, &data, sizeof(data));           /* 读取数据 */
    
```

第 4 行使用 open 函数打开“/dev/xxx_dev”设备文件的时候添加了参数“O_NONBLOCK”, 表示以非阻塞方式打开设备, 这样从设备中读取数据的时候就是非阻塞方式的了。

16.1.2 等待队列

1、等待队列头

阻塞访问最大的好处就是当设备文件不可操作的时候进程可以进入休眠态, 这样可以将 CPU 资源让出来。但是, 当设备文件可以操作的时候就必须唤醒进程, 一般在中断函数里面完成唤醒工作。Linux 内核提供了等待队列(wait queue)来实现阻塞进程的唤醒工作, 如果我们要在驱动中使用等待队列, 必须创建并初始化一个等待队列头, 等待队列头使用结构体 wait_queue_head 表示, wait_queue_head 结构体定义在文件 include/linux/wait.h 中, 结构体内容如下所示:

示例代码 16.1.2.1 wait_queue_head 结构体

```

34 struct wait_queue_head {
35     spinlock_t      lock;
36     struct list_head head;
37 };
38 typedef struct wait_queue_head wait_queue_head_t;
    
```

有些资料里面用 wait_queue_head_t 表示等待队列头, 从 38 行可以看出, wait_queue_head_t 是 wait_queue_head 的别名, 这么做的目的是为了兼容老版本代码。最新版本的系统都用 wait_queue_head 表示等待队列头, 如果你的代码要考虑移植到老版本 linux 内核中, 那么最好使用 wait_queue_head_t 表示等待队列头。

定义好等待队列头以后需要初始化, 使用 init_waitqueue_head 函数初始化等待队列头, 函数原型如下:

```
void init_waitqueue_head(struct wait_queue_head *wq_head)
```

参数 wq_head 就是要初始化的等待队列头。

也可以使用宏 DECLARE_WAIT_QUEUE_HEAD 来一次性完成等待队列头的定义的初始化。

2、等待队列项

等待队列头就是一个等待队列的头部, 每个访问设备的进程都是一个队列项, 当设备不可用的时候就要将这些进程对应的队列项添加到等待队列里面。在以前的 linux 版本中使用结构

体 `wait_queue_t` 表示等待队列项, 在新版本的 linux 内核中已经删除了 `wait_queue_t`, 取而代之的是 `wait_queue_entry` 结构体(其实只是换了个名字, 结构体内的成员变量还是一样的), `wait_queue_entry` 结构体内容如下:

示例代码 16.1.2.2 `wait_queue_entry` 结构体

```
27 struct wait_queue_entry {
28     unsigned int     flags;
29     void             *private;
30     wait_queue_func_t func;
31     struct list_head entry;
32 };
```

使用宏 `DECLARE_WAITQUEUE` 定义并初始化一个等待队列项, 宏的内容如下:

```
DECLARE_WAITQUEUE(name, tsk)
```

`name` 就是等待队列项的名字, `tsk` 表示这个等待队列项属于哪个任务(进程), 一般设置为 `current`, 在 Linux 内核中 `current` 相当于一个全局变量, 表示当前进程。因此宏 `DECLARE_WAITQUEUE` 就是给当前正在运行的进程创建并初始化了一个等待队列项。

3、将队列项添加/移除等待队列头

当设备不可访问的时候就需要将进程对应的等待队列项添加到前面创建的等待队列头中, 只有添加到等待队列头中以后进程才能进入休眠态。当设备可以访问以后再进程对应的等待队列项从等待队列头中移除即可, 等待队列项添加 API 函数如下:

```
void add_wait_queue(struct wait_queue_head *wq_head,
                   struct wait_queue_entry *wq_entry)
```

函数参数和返回值含义如下:

wq_head: 等待队列项要加入的等待队列头。

wq_entry: 要加入的等待队列项。

返回值: 无。

等待队列项移除 API 函数如下:

```
void remove_wait_queue(struct wait_queue_head *wq_head,
                      struct wait_queue_entry *wq_entry)
```

函数参数和返回值含义如下:

wq_head: 要删除的等待队列项所处的等待队列头。

wq_entry: 要删除的等待队列项。

返回值: 无。

4、等待唤醒

当设备可以使用的时候就要唤醒进入休眠态的进程, 唤醒可以使用如下两个函数:

```
void wake_up(struct wait_queue_head *wq_head)
void wake_up_interruptible(struct wait_queue_head *wq_head)
```

参数 `wq_head` 就是要唤醒的等待队列头, 这两个函数会将这个等待队列头中的所有进程都唤醒。`wake_up` 函数可以唤醒处于 `TASK_INTERRUPTIBLE` 和 `TASK_UNINTERRUPTIBLE` 状态的进程, 而 `wake_up_interruptible` 函数只能唤醒处于 `TASK_INTERRUPTIBLE` 状态的进程。

5、等待事件

除了主动唤醒以外, 也可以设置等待队列等待某个事件, 当这个事件满足以后就自动唤醒等待队列中的进程, 和等待事件有关的 API 函数如表 16.1.2.1 所示:

函数	描述
<code>wait_event(wq_head, condition)</code>	等待以 <code>wq_head</code> 为等待队列头的等待队列被唤醒, 前提是 <code>condition</code> 条件必须满足(为真), 否则一直阻塞。此函数会将进程设置为 <code>TASK_UNINTERRUPTIBLE</code> 状态
<code>wait_event_timeout(wq_head, condition, timeout)</code>	功能和 <code>wait_event</code> 类似, 但是此函数可以添加超时时间, 以 <code>jiffies</code> 为单位。此函数有返回值, 如果返回 0 的话表示超时时间到, 而且 <code>condition</code> 为假。为 1 的话表示 <code>condition</code> 为真, 也就是条件满足了。
<code>wait_event_interruptible(wq_head, condition)</code>	与 <code>wait_event</code> 函数类似, 但是此函数将进程设置为 <code>TASK_INTERRUPTIBLE</code> , 就是可以被信号打断。
<code>wait_event_interruptible_timeout(wq_head, condition, timeout)</code>	与 <code>wait_event_timeout</code> 函数类似, 此函数也将进程设置为 <code>TASK_INTERRUPTIBLE</code> , 可以被信号打断。

表 16.1.2.1 等待事件 API 函数

16.1.3 轮询

如果用户应用程序以非阻塞的方式访问设备, 设备驱动程序就要提供非阻塞的处理方式, 也就是轮询。poll、epoll 和 select 可以用于处理轮询, 应用程序通过 select、epoll 或 poll 函数来查询设备是否可以操作, 如果可以操作的话就从设备读取或者向设备写入数据。当应用程序调用 select、epoll 或 poll 函数的时候设备驱动程序中的 poll 函数就会执行, 因此需要在设备驱动程序中编写 poll 函数。我们先来看一下应用程序中使用的 select、poll 和 epoll 这三个函数。

1、select 函数

select 函数原型如下:

```
int select(int nfd,
           fd_set *readfds,
           fd_set *writefds,
           fd_set *exceptfds,
           struct timeval *timeout)
```

函数参数和返回值含义如下:

nfd: 所要监视的这三类文件描述符集合中, 最大文件描述符加 1。

readfds、writefds 和 exceptfds: 这三个指针指向描述符集合, 这三个参数指明了关心哪些描述符、需要满足哪些条件等等, 这三个参数都是 fd_set 类型的, fd_set 类型变量的每一个位都代表了一个文件描述符。readfds 用于监视指定描述符集的读变化, 也就是监视这些文件是否可以读取, 只要这些集合里面有一个文件可以读取那么 select 就会返回一个大于 0 的值表示文件可以读取。如果没有文件可以读取, 那么就会根据 timeout 参数来判断是否超时。可以将 readfds 设置为 NULL, 表示不关心任何文件的读变化。writefds 和 readfds 类似, 只是 writefds 用于监视这些文件是否可以写操作。exceptfds 用于监视这些文件的异常。

比如我们现在要从一个设备文件中读取数据，那么就可以定义一个 `fd_set` 变量，这个变量要传递给参数 `readfds`。当我们定义好一个 `fd_set` 变量以后可以使用如下所示几个宏进行操作：

```
void FD_ZERO(fd_set *set)
void FD_SET(int fd, fd_set *set)
void FD_CLR(int fd, fd_set *set)
int  FD_ISSET(int fd, fd_set *set)
```

`FD_ZERO` 用于将 `fd_set` 变量的所有位都清零，`FD_SET` 用于将 `fd_set` 变量的某个位置 1，也就是向 `fd_set` 添加一个文件描述符，参数 `fd` 就是要加入的文件描述符。`FD_CLR` 用户将 `fd_set` 变量的某个位清零，也就是将一个文件描述符从 `fd_set` 中删除，参数 `fd` 就是要删除的文件描述符。`FD_ISSET` 用于测试一个文件是否属于某个集合，参数 `fd` 就是要判断的文件描述符。

timeout: 超时时间，当我们调用 `select` 函数等待某些文件描述符可以设置超时时间，超时时间使用结构体 `timeval` 表示，结构体定义如下所示：

```
struct timeval {
    long    tv_sec;        /* 秒 */
    long    tv_usec;     /* 微妙 */
};
```

当 `timeout` 为 `NULL` 的时候就表示无限期的等待。

返回值： 0，表示的话就表示超时发生，但是没有任何文件描述符可以进行操作；-1，发生错误；其他值，可以进行操作的文件描述符个数。

使用 `select` 函数对某个设备驱动文件进行读非阻塞访问的操作示例如下所示：

示例代码 16.1.3.1 `select` 函数非阻塞读访问示例

```
1 void main(void)
2 {
3     int ret, fd;                /* 要监视的文件描述符 */
4     fd_set readfds;            /* 读操作文件描述符集 */
5     struct timeval timeout;    /* 超时结构体 */
6
7     fd = open("dev_xxx", O_RDWR | O_NONBLOCK); /* 非阻塞式访问 */
8
9     FD_ZERO(&readfds);        /* 清除 readfds */
10    FD_SET(fd, &readfds);     /* 将 fd 添加到 readfds 里面 */
11
12    /* 构造超时时间 */
13    timeout.tv_sec = 0;
14    timeout.tv_usec = 500000; /* 500ms */
15
16    ret = select(fd + 1, &readfds, NULL, NULL, &timeout);
17    switch (ret) {
18        case 0:                /* 超时 */
19            printf("timeout!\r\n");
20            break;
21        case -1:               /* 错误 */
22            printf("error!\r\n");
```

```

23         break;
24     default:        /* 可以读取数据 */
25         if(FD_ISSET(fd, &readfds)) { /* 判断是否为 fd 文件描述符 */
26             /* 使用 read 函数读取数据 */
27         }
28         break;
29     }
30 }
    
```

2、poll 函数

在单个线程中，select 函数能够监视的文件描述符数量有最大的限制，一般为 1024，可以修改内核将监视的文件描述符数量改大，但是这样会降低效率！这个时候就可以使用 poll 函数，poll 函数本质上和 select 没有太大的差别，但是 poll 函数没有最大文件描述符限制，Linux 应用程序中 poll 函数原型如下所示：

```

int poll(struct pollfd *fds,
         nfd_t         nfd,
         int           timeout)
    
```

函数参数和返回值含义如下：

fds: 要监视的文件描述符集合以及要监视的事件,为一个数组，数组元素都是结构体 pollfd 类型的，pollfd 结构体如下所示：

```

struct pollfd {
    int    fd;        /* 文件描述符 */
    short events;     /* 请求的事件 */
    short revents;    /* 返回的事件 */
};
    
```

fd 是要监视的文件描述符，如果 fd 无效的话那么 events 监视事件也就无效，并且 revents 返回 0。events 是要监视的事件，可监视的事件类型如下所示：

POLLIN	有数据可以读取。
POLLPRI	有紧急的数据需要读取。
POLLOUT	可以写数据。
POLLERR	指定的文件描述符发生错误。
POLLHUP	指定的文件描述符挂起。
POLLNVAL	无效的请求。
POLLRDNORM	等同于 POLLIN

revents 是返回参数，也就是返回的事件，由 Linux 内核设置具体的返回事件。

nfd: poll 函数要监视的文件描述符数量。

timeout: 超时时间，单位为 ms。

返回值：返回 revents 域中不为 0 的 pollfd 结构体个数，也就是发生事件或错误的文件描述符数量；0，超时；-1，发生错误，并且设置 errno 为错误类型。

使用 poll 函数对某个设备驱动文件进行读非阻塞访问的操作示例如下所示：

示例代码 16.1.3.2 poll 函数读非阻塞访问示例

```

1 void main(void)
2 {
3     int ret;
    
```

```

4     int fd;                                /* 要监视的文件描述符 */
5     struct pollfd fds;
6
7     fd = open(filename, O_RDWR | O_NONBLOCK); /* 非阻塞式访问 */
8
9     /* 构造结构体 */
10    fds.fd = fd;
11    fds.events = POLLIN; /* 监视数据是否可以读取 */
12
13    ret = poll(&fds, 1, 500); /* 轮询文件是否可操作, 超时 500ms */
14    if (ret) { /* 数据有效 */
15        .....
16        /* 读取数据 */
17        .....
18    } else if (ret == 0) { /* 超时 */
19        .....
20    } else if (ret < 0) { /* 错误 */
21        .....
22    }
23 }
    
```

3、epoll 函数

传统的 `select` 和 `poll` 函数都会随着所监听的 `fd` 数量的增加, 出现效率低下的问题, 而且 `poll` 函数每次必须遍历所有的描述符来检查就绪的描述符, 这个过程很浪费时间。为此, `epoll` 应运而生, `epoll` 就是为处理大并发而准备的, 一般常常在网络编程中使用 `epoll` 函数。应用程序需要先使用 `epoll_create` 函数创建一个 `epoll` 句柄, `epoll_create` 函数原型如下:

```
int epoll_create(int size)
```

函数参数和返回值含义如下:

size: 从 Linux2.6.8 开始此参数已经没有意义了, 随便填写一个大于 0 的值就可以。

返回值: `epoll` 句柄, 如果为-1 的话表示创建失败。

`epoll` 句柄创建成功以后使用 `epoll_ctl` 函数向其中添加要监视的文件描述符以及监视的事件, `epoll_ctl` 函数原型如下所示:

```
int epoll_ctl(int          epfd,
              int          op,
              int          fd,
              struct epoll_event *event)
```

函数参数和返回值含义如下:

epfd: 要操作的 `epoll` 句柄, 也就是使用 `epoll_create` 函数创建的 `epoll` 句柄。

op: 表示要对 `epfd`(`epoll` 句柄)进行的操作, 可以设置为:

`EPOLL_CTL_ADD` 向 `epfd` 添加文件参数 `fd` 表示的描述符。

`EPOLL_CTL_MOD` 修改参数 `fd` 的 `event` 事件。

`EPOLL_CTL_DEL` 从 `epfd` 中删除 `fd` 描述符。

fd: 要监视的文件描述符。

event: 要监视的事件类型, 为 `epoll_event` 结构体类型指针, `epoll_event` 结构体类型如下所示:

```
struct epoll_event {
    uint32_t    events;      /* epoll 事件    */
    epoll_data_t data;      /* 用户数据    */
};
```

结构体 `epoll_event` 的 `events` 成员变量表示要监视的事件, 可选的事件如下所示:

`EPOLLIN` 有数据可以读取。

`EPOLLOUT` 可以写数据。

`EPOLLPRI` 有紧急的数据需要读取。

`EPOLLERR` 指定的文件描述符发生错误。

`EPOLLHUP` 指定的文件描述符挂起。

`EPOLLET` 设置 `epoll` 为边沿触发, 默认触发模式为水平触发。

`EPOLLONESHOT` 一次性的监视, 当监视完成以后还需要再次监视某个 `fd`, 那么就需要将 `fd` 重新添加到 `epoll` 里面。

上面这些事件可以进行“或”操作, 也就是说可以设置监视多个事件。

返回值: 0, 成功; -1, 失败, 并且设置 `errno` 的值为相应的错误码。

一切都设置好以后应用程序就可以通过 `epoll_wait` 函数来等待事件的发生, 类似 `select` 函数。`epoll_wait` 函数原型如下所示:

```
int epoll_wait(int          epfd,
               struct epoll_event *events,
               int          maxevents,
               int          timeout)
```

函数参数和返回值含义如下:

epfd: 要等待的 `epoll`。

events: 指向 `epoll_event` 结构体的数组, 当有事件发生的时候 Linux 内核会填写 `events`, 调用者可以根据 `events` 判断发生了哪些事件。

maxevents: `events` 数组大小, 必须大于 0。

timeout: 超时时间, 单位为 ms。

返回值: 0, 超时; -1, 错误; 其他值, 准备就绪的文件描述符数量。

`epoll` 更多的是用在大规模的并发服务器上, 因为在这种场合下 `select` 和 `poll` 并不适合。当设计到的文件描述符(`fd`)比较少的时候就适合用 `select` 和 `poll`, 本章我们就使用 `select` 和 `poll` 这两个函数。

16.1.4 Linux 驱动下的 poll 操作函数

当应用程序调用 `select` 或 `poll` 函数来对驱动程序进行非阻塞访问的时候, 驱动程序 `file_operations` 操作集中的 `poll` 函数就会执行。所以驱动程序的编写者需要提供对应的 `poll` 函数, `poll` 函数原型如下所示:

```
unsigned int (*poll) (struct file *filp, struct poll_table_struct *wait)
```

函数参数和返回值含义如下:

filp: 要打开的设备文件(文件描述符)。

wait: 结构体 `poll_table_struct` 类型指针, 由应用程序传递进来的。一般将此参数传递给 `poll_wait` 函数。

返回值:向应用程序返回设备或者资源状态，可以返回的资源状态如下：

POLLIN	有数据可以读取。
POLLPRI	有紧急的数据需要读取。
POLLOUT	可以写数据。
POLLERR	指定的文件描述符发生错误。
POLLHUP	指定的文件描述符挂起。
POLLNVAL	无效的请求。
POLLRDNORM	等同于 POLLIN，普通数据可读

我们需要在驱动程序的 poll 函数中调用 poll_wait 函数，poll_wait 函数不会引起阻塞，只是将应用程序添加到 poll_table 中，poll_wait 函数原型如下：

```
void poll_wait(struct file * filp, wait_queue_head_t * wait_address, poll_table *p)
```

参数 wait_address 是要添加到 poll_table 中的等待队列头，参数 p 就是 poll_table，就是 file_operations 中 poll 函数的 wait 参数。

16.2 阻塞 IO 实验

在上一章 Linux 中断实验中，我们直接在应用程序中通过 read 函数不断的读取按键状态，当按键有效的时候就打印出按键值。这种方法有个缺点，那就是 keyirqApp 这个测试应用程序拥有很高的 CPU 占用率，大家可以在开发板中加载上一章的驱动程序模块 keyirq.ko，然后以后台运行模式打开 keyirqApp 这个测试软件，命令如下：

```
./keyirqApp /dev/keyirq &
```

测试驱动是否正常工作，如果驱动工作正常的话输入“top”命令，top 命令打开资源界面以后输入大写“P”，按照 CPU 使用率从高到低排序。可以看到 keyirqApp 这个应用程序的 CPU 使用率，结果如图 16.2.1 所示：

PID	USER	PR	NI	VIRT	RES	%CPU	%MEM	TIME+	S	COMMAND
870	root	20	0	1.7m	0.2m	100.0	0.0	0:51.86	R	./keyirq+
871	root	20	0	9.0m	2.3m	2.0	0.1	0:01.19	R	top
2	root	20	0	0.0m	0.0m	0.0	0.0	0:00.01	S	[kthreadd]
3	root	0	-20	0.0m	0.0m	0.0	0.0	0:00.00	I	[rcu gp]

图 16.2.1 CPU 使用率

从图 16.2.1 可以看出，keyirqApp 这个应用程序的 CPU 使用率竟然高达 100%，这仅仅是一个读取按键值的应用程序，这么高的 CPU 使用率显然是有问题的！原因就在于我们是直接在 while 循环中通过 read 函数读取按键值，因此 keyirqApp 这个软件会一直运行，一直读取按键值，CPU 使用率肯定就会很高。最好的方法就是在没有有效的按键事件发生的时候，keyirqApp 这个应用程序应该处于休眠状态，当有按键事件发生以后 keyirqApp 这个应用程序才运行，打印出按键值，这样就会降低 CPU 使用率，本小节我们就使用阻塞 IO 来实现此能。

16.2.1 硬件原理图分析

本章实验硬件原理图参考 13.2 小节即可。

16.2.2 实验程序编写

1、驱动程序编写

本实验对应的例程路径为：[开发板光盘](#)→01、[程序源码](#)→Linux 驱动例程→13_blockio。

本章实验我们在上一章的“12_irq”实验的基础上完成,主要是对其添加阻塞访问相关的代码。新建名为“13_blockio”的文件夹,然后在 13_blockio 文件夹里面创建 vscode 工程,工作区命名为“blockio”。将“12_irq”实验中的 keyirq.c 复制到 13_blockio 文件夹中,并重命名为 blockio.c。接下来我们就修改 blockio.c 这个文件,在其中添加阻塞相关的代码,完成以后的 blockio.c 内容如下所示(因为是在上一章实验的 keyirq.c 文件的基础上修改的,为了减少篇幅,下面的代码有省略):

示例代码 16.2.2.1 blockio.c 文件代码(有省略)

```

41  /* key 设备结构体 */
42  struct key_dev{
43      dev_t devid;                /* 设备号      */
44      struct cdev cdev;          /* cdev        */
45      struct class *class;       /* 类          */
46      struct device *device;     /* 设备        */
47      struct device_node *nd;    /* 设备节点    */
48      int key_gpio;              /* key 所使用的 GPIO 编号  */
49      struct timer_list timer;   /* 按键值      */
50      int irq_num;               /* 中断号      */
51
52      atomic_t status;           /* 按键状态    */
53      wait_queue_head_t r_wait; /* 读等待队列头 */
54  };
55
56  .....
150 static void key_timer_function(struct timer_list *arg)
151 {
152     static int last_val = 0;
153     int current_val;
154
155     /* 读取按键值并判断按键当前状态 */
156     current_val = gpio_get_value(key.key_gpio);
157     if (1 == current_val && !last_val){
158         atomic_set(&key.status, KEY_PRESS); /* 按下 */
159         wake_up_interruptible(&key.r_wait);
160     }
161     else if (0 == current_val && last_val) {
162         atomic_set(&key.status, KEY_RELEASE); /* 松开 */
163         wake_up_interruptible(&key.r_wait);
164     }
165     else
166         atomic_set(&key.status, KEY_KEEP); /* 状态保持 */
167
168     last_val = current_val;
169 }
170
    
```

```

171 /*
172 * @description   : 打开设备
173 * @param - inode : 传递给驱动的 inode
174 * @param - filp  : 设备文件, file 结构体有个叫做 private_data 的成员变量
175 *                  一般在 open 的时候将 private_data 指向设备结构体。
176 * @return        : 0 成功;其他 失败
177 */
178 static int key_open(struct inode *inode, struct file *filp)
179 {
180     return 0;
181 }
182
183 /*
184 * @description   : 从设备读取数据
185 * @param - filp  : 要打开的设备文件 (文件描述符)
186 * @param - buf   : 返回给用户空间的数据缓冲区
187 * @param - cnt   : 要读取的数据长度
188 * @param - offt  : 相对于文件首地址的偏移
189 * @return        : 读取的字节数, 如果为负值, 表示读取失败
190 */
191 static ssize_t key_read(struct file *filp, char __user *buf,
192                        size_t cnt, loff_t *offt)
193 {
194     int ret;
195
196     /* 加入等待队列, 当有按键按下或松开动作发生时, 才会被唤醒 */
197     ret = wait_event_interruptible(key.r_wait, KEY_KEEP !=
198                                   atomic_read(&key.status));
199
200     if(ret)
201         return ret;
202
203     /* 将按键状态信息发送给应用程序 */
204     ret = copy_to_user(buf, &key.status, sizeof(int));
205
206     /* 状态重置 */
207     atomic_set(&key.status, KEY_KEEP);
208
209     return ret;
210 }
211 .....
212 /*
213 * @description   : 驱动入口函数
214 * @param        : 无

```

```

245 * @return      : 无
246 */
247 static int __init mykey_init(void)
248 {
249     int ret;
250
251     /* 初始化等待队列头 */
252     init_waitqueue_head(&key.r_wait);
253
254     /* 初始化 timer, 设置定时器处理函数, 还未设置周期, 所有不会激活定时器 */
255     timer_setup(&key.timer, key_timer_function, 0);
256
257     /* 初始化按键状态 */
258     atomic_set(&key.status, KEY_KEEP);
259
260     .....
311 }
261
262 .....
    
```

第 42~54 行, 我们删除了设备结构体 `struct key_dev` 中的自旋锁变量, 本章驱动代码我们不用自旋锁, 而改成第 52 行的原子变量来实现对相应变量的保护操作。第 53 行添加了一个等待队列头 `r_wait`, 因为在 Linux 驱动中处理阻塞 IO 需要用到等待队列, 而等待队列会使进程进入休眠状态, 所以不能使用自旋锁!

第 150~169 行, 定时器定时处理函数 `key_timer_function`, 对按键状态进行判断, 如果是按下动作或是松开动作则会使用 `wake_up_interruptible` 函数唤醒等待队列, 并且将 `status` 变量设置为 `KEY_PRESS` 或 `KEY_RELEASE`; 这样在 `key_read` 函数中阻塞的进程就会解除阻塞继续进行下面的操作。

第 191~208 行, `key_read` 函数, 在这个函数中我们会判断按键是否有按下或松开动作发生时, 如果没有则调用 `wait_event_interruptible` 把它加入等待队列当中, 进行阻塞。如果等待队列被唤醒并且条件 “`KEY_KEEP != atomic_read(&status)`” 成立, 则解除阻塞, 继续下面的操作, 也就是读取按键状态数据将其发送给应用程序。因为采用了 `wait_event_interruptible` 函数, 因此进入休眠态的进程可以被信号打断。在该函数中我们读取 `status` 原子变量必须要使用 `atomic_read` 函数进行操作。

第 252 行, 在驱动入口函数中我们会调用 `init_waitqueue_head` 初始化等待队列头; 第 258 行使用原子操作 `atomic_set` 设置按键的初始状态 `status` 为 `KEY_KEEP`。

使用等待队列实现阻塞访问重点注意两点:

- ①、将任务或者进程加入到等待队列头,
- ②、在合适的点唤醒等待队列, 一般是中断处理函数里面。

2、编写测试 APP

本节实验的测试 APP 直接使用 `12_irq` 实验目录下的 `keyirqApp.c` 测试程序, 将 `keyirqApp.c` 复制到本实验目录 `13_blockio` 中, 重命名为 `blockApp.c`, 不需要修改任何内容。

16.2.3 运行测试

1、编译驱动程序和测试 APP

①、编译驱动程序

编写 Makefile 文件，本章实验的 Makefile 文件和第五章实验基本一样，只是将 obj-m 变量的值改为 blockio.o，Makefile 内容如下所示：

示例代码 16.2.3.1 Makefile 文件

```
1 KERNELDIR := /home/alientek/rk3568_linux_sdk/kernel
.....
4 obj-m := blockio.o
.....
11 clean:
12 $(MAKE) -C $(KERNELDIR) M=$(CURRENT_PATH) clean
```

第 4 行，设置 obj-m 变量的值为 blockio.o。

输入如下命令编译出驱动模块文件：

```
make ARCH=arm64 //ARCH=arm64 必须指定，否则编译会失败
```

编译成功以后就会生成一个名为“blockio.ko”的驱动模块文件。

②、编译测试 APP

输入如下命令编译测试 blockioApp.c 这个测试程序：

```
/opt/atk-dlrk356x-toolchain/bin/aarch64-buildroot-linux-gnu-gcc blockioApp.c -o blockioApp
```

编译成功以后就会生成 blockioApp 这个应用程序。

2、运行测试

在 Ubuntu 中将上一小节编译出来的 blockio.ko 和 blockioApp 这两个文件通过 adb 命令发送到开发板的 /lib/modules/4.19.232 目录下，命令如下：

```
adb push blockio.ko blockioApp /lib/modules/4.19.232
```

发送成功以后进入到开发板目录 lib/modules/4.19.232 中，输入如下命令加载 blockio.ko 驱动模块：

```
depmod //第一次加载驱动的时候需要运行此命令
```

```
modprobe blockio //加载驱动
```

驱动加载成功以后使用如下命令打开 blockioApp 这个测试 APP，并且以后台模式运行：

```
./blockioApp /dev/key &
```

使用杜邦线将图 13.2.1 中 GPIO3_C5 这个 IO 接到开发板的 3.3V 电压上，模拟按键被按下，结果如图 16.2.3.1 所示：

```
root@ATK-DLRK356X:/lib/modules/4.19.232# ./blockioApp /dev/key &
[1] 867
root@ATK-DLRK356X:/lib/modules/4.19.232# Key Press
Key Release
Key Press
Key Release
Key Press
Key Release
Key Press
Key Release
```

图 16.2.3.1 测试 APP 运行测试

首先输入“ps -aux”命令查看 blockioApp 这个进程的 PID，比如在我的开发板上如图 16.2.3.2 所示：

root	803	0.0	0.0	2528	316	?	S	17:35	0:00	/usr/sbin/phpc2s
root	809	0.0	0.0	1940	96	?	Ss	17:35	0:00	input-event-dae
root	810	0.0	0.0	9372	3084	ttyFIQ0	Ss	17:35	0:00	~/bin/sh
root	867	0.0	0.0	1908	192	ttyFIQ0	S	17:38	0:00	./blockioApp /d
root	876	0.0	0.0	0	0	?	I	17:40	0:00	[kworker/u8:3-e

图 16.2.3.2 图 16.2.3.2 blockioApp 的 PID

从图 16.2.3.2 可以看出，blockioApp 的 PID 为 972，我们可以输入：

```
top -p 867 //查看 PID 为 867 的进程资源信息
```

上面就是使用 top 命令只查看 PID 为 972 这个进程的资源消耗情况，如图 16.2.3.3 所示：

PID	USER	PR	NI	VIRT	RES	%CPU	%MEM	TIME+	S	COMMAND
867	root	20	0	1.9m	0.2m	0.0	0.0	0:00.00	S	./blockioApp /de+

图 16.2.3.3 应用程序 CPU 使用率

从图 16.2.3.3 可以看出，当我们在按键驱动程序里面加入阻塞访问以后，blockioApp 这个应用程序的 CPU 使用率从图 16.2.1 中的 100%降低到了 0%。大家注意，这里的 0%并不是说 blockioApp 这个应用程序不使用 CPU 了，只是因为使用率太小了，CPU 使用率可能为 0.00001%，但是图 16.2.3.3 是没有小数点显示的，因此就显示成了 0%。

我们可以使用“kill”命令关闭后台运行的应用程序，比如我们关闭掉 blockioApp 这个后台运行的应用程序。首先输出“Ctrl+C”关闭 top 命令界面，进入到命令行模式。然后使用“ps -aux”命令查看一下 blockioApp 这个应用程序的 PID，这个前面已经做了，也就是图 16.2.3.2 中的 972。

使用“kill -9 PID”即可“杀死”指定 PID 的进程，比如我们现在要“杀死”PID 为 972 的 blockioApp 应用程序，可是使用如下命令：

```
kill -9 867
```

输入上述命令以后终端显示如图 16.2.3.4 所示：

```
root@ATK-DLRK356X:/lib/modules/4.19.232# kill -9 867
root@ATK-DLRK356X:/lib/modules/4.19.232#
[1]+  Killed                  ./blockioApp /dev/key
root@ATK-DLRK356X:/lib/modules/4.19.232#
```

图 16.2.3.4 kill 命令输出结果

从图 16.2.3.4 可以看出，“./blockioApp /dev/key”这个应用程序已经被“杀掉”了，在此输入“ps”命令查看当前系统运行的进程，会发现 blockioApp 已经不见了。这个就是使用 kill 命令“杀掉”指定进程的方法。

16.3 非阻塞 IO 实验

16.3.1 硬件原理图分析

本章实验硬件原理图参考 13.2 小节即可。

16.3.2 实验程序编写

1、驱动程序编写

本实验对应的例程路径为：**开发板光盘**→**01、程序源码**→**Linux 驱动例程**→**14_noblockio**。

本章实验我们在 16.2 小节中的“13_blockio”实验的基础上完成，上一小节实验我们已经在驱动中添加了阻塞 IO 的代码，本小节我们继续完善驱动，加入非阻塞 IO 驱动代码。新建名

为“14_noblockio”的文件夹，然后在 14_noblockio 文件夹里面创建 vscode 工程，工作区命名为“noblockio”。将“13_blockio”实验中的 blockio.c 复制到 14_noblockio 文件夹中，并重命名为 noblockio.c。接下来我们就修改 noblockio.c 这个文件，在其中添加非阻塞相关的代码，完成以后的 noblockio.c 内容如下所示(因为是在上一小节实验的 blockio.c 文件的基础上修改的，为了减少篇幅，下面的代码有省略):

示例代码 16.3.3.1 noblockio.c 文件(有省略)

```

.....
27 #include <linux/wait.h>
28 #include <linux/poll.h>
29 // #include <asm/mach/map.h>
30 #include <asm/uaccess.h>
31 #include <asm/io.h>
32
.....
185 /*
186  * @description   : 从设备读取数据
187  * @param - filp  : 要打开的设备文件(文件描述符)
188  * @param - buf   : 返回给用户空间的数据缓冲区
189  * @param - cnt   : 要读取的数据长度
190  * @param - offt  : 相对于文件首地址的偏移
191  * @return        : 读取的字节数，如果为负值，表示读取失败
192  */
193 static ssize_t key_read(struct file *filp, char __user *buf,
194                        size_t cnt, loff_t *offt)
195 {
196     int ret;
197
198     if (filp->f_flags & O_NONBLOCK) { /*非阻塞方式访问 */
199         if(KEY_KEEP == atomic_read(&key.status))
200             return -EAGAIN;
201     } else { /* 阻塞方式访问 */
202         /* 加入等待队列，当有按键按下或松开动作发生时，才会被唤醒 */
203         ret = wait_event_interruptible(key.r_wait, KEY_KEEP !=
204                                     atomic_read(&key.status));
205
206         if(ret)
207             return ret;
208     }
209     /* 将按键状态信息发送给应用程序 */
210     ret = copy_to_user(buf, &key.status, sizeof(int));
211
212     /* 状态重置 */
213     atomic_set(&key.status, KEY_KEEP);
214

```

```

213     return ret;
214 }
.....
239 /*
240  * @description   : poll 函数, 用于处理非阻塞访问
241  * @param - filp  : 要打开的设备文件(文件描述符)
242  * @param - wait  : 等待列表(poll_table)
243  * @return        : 设备或者资源状态,
244  */
245 static unsigned int key_poll(struct file *filp,
                               struct poll_table_struct *wait)
246 {
247     unsigned int mask = 0;
248
249     poll_wait(filp, &key.r_wait, wait);
250
251     if(KEY_KEEP != atomic_read(&key.status)) /* 按键按下或松开动作发生 */
252         mask = POLLIN | POLLRDNORM; /* 返回 POLLIN */
253
254     return mask;
255 }
256
257 /* 设备操作函数 */
258 static struct file_operations key_fops = {
259     .owner = THIS_MODULE,
260     .open = key_open,
261     .read = key_read,
262     .write = key_write,
263     .release = key_release,
264     .poll = key_poll,
265 };
.....
    
```

第 28 行, 使用 include 将内核源码目录 include/linux/poll.h 头文件包含进来。

第 193~214 行, key_read 函数中判断是否为非阻塞式读取访问, 如果是的话就判断按键状态是否有效, 也就是判断是否产生了按下或松开这样的动作, 如果没有的话就返回-EAGAIN。

第 245~255 行, key_poll 函数就是 file_operations 驱动操作集中的 poll 函数, 当应用程序调用 select 或者 poll 函数的时候 key_poll 函数就会执行。第 249 行调用 poll_wait 函数将等待队列头添加到 poll_table 中, 第 251~252 行判断按键是否有效, 如果按键有效的话就向应用程序返回 POLLIN 这个事件, 表示有数据可以读取。

第 264 行, 设置 file_operations 的 poll 成员变量为 key_poll。

2、编写测试 APP

新建名为 noblockioApp.c 测试 APP 文件, 然后在其中输入如下所示内容:

示例代码 16.3.3.2 noblockioApp.c 文件代码

```

12 #include <stdio.h>
13 #include <unistd.h>
14 #include <sys/types.h>
15 #include <sys/stat.h>
16 #include <fcntl.h>
17 #include <stdlib.h>
18 #include <string.h>
19 #include <poll.h>
20
21 /*
22  * @description   : main 主程序
23  * @param - argc  : argv 数组元素个数
24  * @param - argv  : 具体参数
25  * @return        : 0 成功;其他 失败
26  */
27 int main(int argc, char *argv[])
28 {
29     fd_set readfds;
30     int key_val;
31     int fd;
32     int ret;
33
34     /* 判断传参个数是否正确 */
35     if(2 != argc) {
36         printf("Usage:\n"
37             "\t./keyApp /dev/key\n"
38             );
39         return -1;
40     }
41
42     /* 打开设备 */
43     fd = open(argv[1], O_RDONLY | O_NONBLOCK);
44     if(0 > fd) {
45         printf("ERROR: %s file open failed!\n", argv[1]);
46         return -1;
47     }
48
49     FD_ZERO(&readfds);
50     FD_SET(fd, &readfds);
51
52     /* 循环轮训读取按键数据 */
53     for ( ; ; ) {
54

```

```

55     ret = select(fd + 1, &readfds, NULL, NULL, NULL);
56     switch (ret) {
57
58     case 0:      /* 超时 */
59         /* 用户自定义超时处理 */
60         break;
61
62     case -1:    /* 错误 */
63         /* 用户自定义错误处理 */
64         break;
65
66     default:
67         if(FD_ISSET(fd, &readfds)) {
68             read(fd, &key_val, sizeof(int));
69             if (0 == key_val)
70                 printf("Key Press\n");
71             else if (1 == key_val)
72                 printf("Key Release\n");
73         }
74
75         break;
76     }
77 }
78
79 /* 关闭设备 */
80 close(fd);
81 return 0;
82 }
    
```

第 53~77 行, 在本测试程序中我们使用 select 函数来实现非阻塞访问, 在 for 循环中使用 select 函数不断的轮询, 检查驱动程序是否有数据可以读取, 如果可以读取的话就调用 read 函数读取按键数据。大家也可以试试使用 poll 函数来实现!

16.3.3 运行测试

1、编译驱动程序和测试 APP

①、编译驱动程序

编写 Makefile 文件, 本章实验的 Makefile 文件和第五章实验基本一样, 只是将 obj-m 变量的值改为 noblockio.o, Makefile 内容如下所示:

```

示例代码 16.3.3.1 Makefile 文件
1  KERNELDIR := /home/alientek/rk3568_linux_sdk/kernel
.....
4  obj-m := noblockio.o
.....
    
```

```
11 clean:
12 $(MAKE) -C $(KERNELDIR) M=$(CURRENT_PATH) clean
```

第 4 行, 设置 obj-m 变量的值为 noblockio.o。

输入如下命令编译出驱动模块文件:

```
make ARCH=arm64 //ARCH=arm64 必须指定, 否则编译会失败
```

编译成功以后就会生成一个名为“noblockio.ko”的驱动模块文件。

②、编译测试 APP

输入如下命令编译测试 noblockioApp.c 这个测试程序:

```
/opt/atk-dlrk356x-toolchain/bin/aarch64-buildroot-linux-gnu-gcc noblockioApp.c -o
```

noblockioApp

编译成功以后就会生成 noblcokioApp 这个应用程序。

2、运行测试

在 Ubuntu 中将上一小节编译出来的 noblockio.ko 和 noblockioApp 这两个文件通过 adb 命令发送到开发板的 /lib/modules/4.19.232 目录下, 命令如下:

```
adb push noblockio.ko noblockioApp /lib/modules/4.19.232
```

发送成功以后进入到开发板目录 lib/modules/4.19.232 中, 输入如下命令加载 noblockio.ko 驱动模块:

```
depmod //第一次加载驱动的时候需要运行此命令
```

```
modprobe noblockio //加载驱动
```

驱动加载成功以后使用如下命令打开 noblockioApp 这个测试 APP, 并且以后台模式运行:

```
./noblockioApp /dev/key &
```

使用杜邦线将图 13.2.1 中 GPIO3_C5 这个 IO 接到开发板的 3.3V 电压上, 模拟按键被按下, 结果如图 16.3.3.1 所示:

```
root@ATK-DLRK356X:/lib/modules/4.19.232# ./noblockioApp /dev/key &
[1] 1029
root@ATK-DLRK356X:/lib/modules/4.19.232# Key Press
Key Release
Key Press
Key Release
Key Press
Key Release
Key Press
Key Release
```

图 16.3.3.1 测试 APP 运行测试

当 GPIO3_C5 接到 3.3V 以后, 模拟按键按下, noblockioApp 这个测试 APP 就会打印出按键信息。使用 top 命令查看 noblockioAPP 这个应用 APP 的 CPU 使用率, 如图 16.3.3.2 所示:

PID	USER	PR	NI	VIRT	RES	%CPU	%MEM	TIME+	S	COMMAND
1004	root	20	0	1.4m	0.2m	0.0	0.0	0:00.00	S	noblockioApp

图 16.3.3.2 应用程序 CPU 使用率

从图 16.3.3.2 可以看出, 采用非阻塞方式读处理以后, noblockioApp 的 CPU 占用率也低至 0%, 和图 16.2.3.3 中的 blockioApp 一样, 这里的 0%并不是说 noblockioApp 这个应用程序不使用 CPU 了, 只是因为使用率太小了。

如果要“杀掉”处于后台运行模式的 noblockioApp 这个应用程序, 可以参考 16.2.3 小节讲解的方法。

第十七章 异步通知实验

在前面使用阻塞或者非阻塞的方式来读取驱动中按键值都是应用程序主动读取的，对于非阻塞方式来说还需要应用程序通过 `poll` 函数不断的轮询。最好的方式就是驱动程序能主动向应用程序发出通知，报告自己可以访问，然后应用程序再从驱动程序中读取或写入数据，类似于中断。`Linux` 提供了异步通知这个机制来完成此功能，本章我们就来学习一下异步通知以及如何在驱动中添加异步通知相关处理代码。

17.1 异步通知

17.1.1 异步通知简介

我们首先来回顾一下“中断”，中断是处理器提供的一种异步机制，我们配置好中断以后就可以让处理器去处理其他的事情了，当中断发生以后会触发我们事先设置好的中断服务函数，在中断服务函数中做具体的处理。比如我们以前学习单片机的时候用到的 GPIO 按键中断，我们通过按键去开关蜂鸣器，采用中断以后处理器就不需要时刻的去查看按键有没有被按下，因为按键按下以后会自动触发中断。同样的，Linux 应用程序可以通过阻塞或者非阻塞这两种方式来访问驱动设备，通过阻塞方式访问的话应用程序会处于休眠态，等待驱动设备可以使用。非阻塞方式的话会通过 poll 函数来不断的轮询，查看驱动设备文件是否可以访问。这两种方式都需要应用程序主动的去查询设备的使用情况，如果能提供一种类似中断的机制，当驱动程序可以访问的时候主动告诉应用程序那就最好了。

“信号”为此应运而生，信号类似于我们硬件上使用的“中断”，只不过信号是软件层次上的。算是在软件层次上对中断的一种模拟，驱动可以通过主动向应用程序发送信号的方式来报告自己可以访问了，应用程序获取到信号以后就可以从驱动设备中读取或者写入数据了。整个过程就相当于应用程序收到了驱动发送过来了的一个中断，然后应用程序去响应这个中断，在整个处理过程中应用程序并没有去查询驱动设备是否可以访问，一切都是由驱动设备自己告诉给应用程序的。

阻塞、非阻塞、异步通知，这三种是针对不同的场合提出来的不同的解决方法，没有优劣之分，在实际的工作和学习中，根据自己的实际需求选择合适的处理方法即可。

异步通知的核心就是信号，在 arch/xtensa/include/uapi/asm/signal.h 文件中定义了 Linux 所支持的所有信号，这些信号如下所示：

示例代码 17.1.1.1 Linux 信号			
18	#define	SIGHUP	1 /* 终端挂起或控制进程终止 */
19	#define	SIGINT	2 /* 终端中断 (Ctrl+C 组合键) */
20	#define	SIGQUIT	3 /* 终端退出 (Ctrl+\ 组合键) */
21	#define	SIGILL	4 /* 非法指令 */
22	#define	SIGTRAP	5 /* debug 使用，有断点指令产生 */
23	#define	SIGABRT	6 /* 由 abort(3) 发出的退出指令 */
24	#define	SIGIOT	6 /* IOT 指令 */
25	#define	SIGBUS	7 /* 总线错误 */
26	#define	SIGFPE	8 /* 浮点运算错误 */
27	#define	SIGKILL	9 /* 杀死、终止进程 */
28	#define	SIGUSR1	10 /* 用户自定义信号 1 */
29	#define	SIGSEGV	11 /* 段违例 (无效的内存段) */
30	#define	SIGUSR2	12 /* 用户自定义信号 2 */
31	#define	SIGPIPE	13 /* 向非读管道写入数据 */
32	#define	SIGALRM	14 /* 闹钟 */
33	#define	SIGTERM	15 /* 软件终止 */
34	#define	SIGSTKFLT	16 /* 栈异常 */
35	#define	SIGCHLD	17 /* 子进程结束 */
36	#define	SIGCONT	18 /* 进程继续 */
37	#define	SIGSTOP	19 /* 停止进程的执行，只是暂停 */

```

38 #define SIGTSTP      20      /* 停止进程的运行 (Ctrl+Z 组合键) */
39 #define SIGTTIN      21      /* 后台进程需要从终端读取数据 */
40 #define SIGTTOU      22      /* 后台进程需要向终端写数据 */
41 #define SIGURG        23      /* 有"紧急"数据 */
42 #define SIGXCPU       24      /* 超过 CPU 资源限制 */
43 #define SIGXFSS       25      /* 文件大小超额 */
44 #define SIGVTALRM     26      /* 虚拟时钟信号 */
45 #define SIGPROF       27      /* 时钟信号描述 */
46 #define SIGWINCH     28      /* 窗口大小改变 */
47 #define SIGIO        29      /* 可以进行输入/输出操作 */
48 #define SIGPOLL       SIGIO
49 /* #define SIGLOS     29 */
50 #define SIGPWR        30      /* 断点重启 */
51 #define SIGSYS        31      /* 非法的系统调用 */
52 #define SIGUNUSED     31      /* 未使用信号 */
    
```

在示例代码 17.1.1.1 中的这些信号中，除了 SIGKILL(9)和 SIGSTOP(19)这两个信号不能被忽略外，其他的信号都可以忽略。这些信号就相当于中断号，不同的中断号代表了不同的中断，不同的中断所做的处理不同，因此，驱动程序可以通过向应用程序发送不同的信号来实现不同的功能。

我们使用中断的时候需要设置中断处理函数，同样的，如果要在应用程序中使用信号，那么就必须设置信号所使用的信号处理函数，在应用程序中使用 `signal` 函数来设置指定信号的处理函数，`signal` 函数原型如下所示：

```
sighandler_t signal(int signum, sighandler_t handler)
```

函数参数和返回值含义如下：

signum：要设置处理函数的信号。

handler：信号的处理函数。

返回值：设置成功的话返回信号的前一个处理函数，设置失败的话返回 SIG_ERR。

信号处理函数原型如下所示：

```
typedef void (*sighandler_t)(int)
```

我们前面讲解的使用“kill -9 PID”杀死指定进程的方法就是向指定的进程(PID)发送 SIGKILL 这个信号。当按下键盘上的 CTRL+C 组合键以后会向当前正在占用终端的应用程序发出 SIGINT 信号，SIGINT 信号默认的动作是关闭当前应用程序。这里我们修改一下 SIGINT 信号的默认处理函数，当按下 CTRL+C 组合键以后先在终端上打印出“SIGINT signal!”这行字符串，然后再关闭当前应用程序。新建 `signaltest.c` 文件，然后输入如下所示内容：

示例代码 17.1.1.2 信号测试

```

1 #include <stdlib.h>
2 #include <stdio.h>
3 #include <signal.h>
4
5 void sigint_handler(int num)
6 {
7     printf("\r\nSIGINT signal!\r\n");
8     exit(0);
    
```

```

9 }
10
11 int main(void)
12 {
13     signal(SIGINT, sigint_handler);
14     while(1);
15     return 0;
16 }
    
```

在示例代码 17.1.1.2 中我们设置 SIGINT 信号的处理函数为 sigint_handler, 当按下 CTRL+C 向 signaltest 发送 SIGINT 信号以后 sigint_handler 函数就会执行, 此函数先输出一行 “SIGINT signal!” 字符串, 然后调用 exit 函数关闭 signaltest 应用程序。

使用如下命令编译 signaltest.c:

```
gcc signaltest.c -o signaltest
```

然后输入 “./signaltest” 命令打开 signaltest 这个应用程序, 然后按下键盘上的 CTRL+C 组合键, 结果如图 17.1.1.1 所示:

```

alientek@ubuntu:~/Linux_Drivers$ ./signaltest
^C
SIGINT signal!
alientek@ubuntu:~/Linux_Drivers$
    
```

图 17.1.1.1 signaltest 软件运行结果

从图 17.1.1.1 可以看出, 当按下 CTRL+C 组合键以后 sigint_handler 这个 SIGINT 信号处理函数执行了, 并且输出了 “SIGINT signal!” 这行字符串。

17.1.2 驱动中的信号处理

1、fasync_struct 结构体

首先我们需要在驱动程序中定义一个 fasync_struct 结构体指针变量, fasync_struct 结构体内容如下:

示例代码 17.1.2.1 fasync_struct 发结构体

```

struct fasync_struct {
    spinlock_t    fa_lock;
    int           magic;
    int           fa_fd;
    struct fasync_struct *fa_next;
    struct file   *fa_file;
    struct rcu_head fa_rcu;
};
    
```

一般将 fasync_struct 结构体指针变量定义到设备结构体中, 比如在上一章节的 key_dev 结构体中添加一个 fasync_struct 结构体指针变量, 结果如下所示:

示例代码 17.1.2.2 在设备结构体中添加 fasync_struct 类型变量指针

```

1 struct key_dev{
2     dev_t      devid;
3     struct cdev cdev;
4     struct class *class;
    
```

.....

```
12     struct fasync_struct *async_queue; /* fasync_struct 结构体 */
13 };
```

第 12 行就是在 imx6uirq_dev 中添加了一个 fasync_struct 结构体指针变量。

2、fasync 函数

如果要使用异步通知，需要在设备驱动中实现 file_operations 操作集中的 fasync 函数，此函数格式如下所示：

```
int (*fasync)(int fd, struct file *filp, int on)
```

fasync 函数里面一般通过调用 fasync_helper 函数来初始化前面定义的 fasync_struct 结构体指针，fasync_helper 函数原型如下：

```
int fasync_helper(int fd, struct file * filp, int on, struct fasync_struct **fapp)
```

fasync_helper 函数的前三个参数就是 fasync 函数的那三个参数，第四个参数就是要初始化的 fasync_struct 结构体指针变量。当应用程序通过“fcntl(fd, F_SETFL, flags | FASYNC)”改变 fasync 标记的时候，驱动程序 file_operations 操作集中的 fasync 函数就会执行。

驱动程序中的 fasync 函数参考示例如下：

示例代码 17.1.2.3 驱动中 fasync 函数参考示例

```
1 struct xxx_dev {
2     .....
3     struct fasync_struct *async_queue; /* 异步相关结构体 */
4 };
5
6 static int xxx_fasync(int fd, struct file *filp, int on)
7 {
8     struct xxx_dev *dev = (xxx_dev)filp->private_data;
9
10    if (fasync_helper(fd, filp, on, &dev->async_queue) < 0)
11        return -EIO;
12    return 0;
13 }
14
15 static struct file_operations xxx_ops = {
16     .....
17     .fasync = xxx_fasync,
18     .....
19 };
```

在关闭驱动文件的时候需要在 file_operations 操作集中的 release 函数中释放 fasync_struct，fasync_struct 的释放函数同样为 fasync_helper，release 函数参数参考实例如下：

示例代码 17.1.2.4 释放 fasync_struct 参考示例

```
1 static int xxx_release(struct inode *inode, struct file *filp)
2 {
3     return xxx_fasync(-1, filp, 0); /* 删除异步通知 */
4 }
5
```

```
6 static struct file_operations xxx_ops = {
7     .....
8     .release = xxx_release,
9 };
```

第 3 行通过调用示例代码 17.1.2.3 中的 `xxx_fasync` 函数来完成 `fasync_struct` 的释放工作，但是，其最终还是通过 `fasync_helper` 函数完成释放工作。

1、kill_fasync 函数

当设备可以访问的时候，驱动程序需要向应用程序发出信号，相当于产生“中断”。`kill_fasync` 函数负责发送指定的信号，`kill_fasync` 函数原型如下所示：

```
void kill_fasync(struct fasync_struct **fp, int sig, int band)
```

函数参数和返回值含义如下：

fp: 要操作的 `fasync_struct`。

sig: 要发送的信号。

band: 可读时设置为 `POLL_IN`，可写时设置为 `POLL_OUT`。

返回值: 无。

17.1.3 应用程序对异步通知的处理

应用程序对异步通知的处理包括以下三步：

1、注册信号处理函数

应用程序根据驱动程序所使用的信号来设置信号的处理函数，应用程序使用 `signal` 函数来设置信号的处理函数。前面已经详细的讲过了，这里就不细讲了。

2、将本应用程序的进程号告诉给内核

使用 `fcntl(fd, F_SETOWN, getpid())` 将本应用程序的进程号告诉给内核。

3、开启异步通知

使用如下两行程序开启异步通知：

```
flags = fcntl(fd, F_GETFL); /* 获取当前的进程状态 */
fcntl(fd, F_SETFL, flags | FASYNC); /* 开启当前进程异步通知功能 */
```

重点就是通过 `fcntl` 函数设置进程状态为 `FASYNC`，经过这一步，驱动程序中的 `fasync` 函数就会执行。

17.2 硬件原理图分析

本章实验硬件原理图参考 13.2 小节即可。

17.3 实验程序编写

本实验对应的例程路径为：[开发板光盘](#)→01、[程序源码](#)→Linux 驱动例程→15_asyncnoti。

本章实验我们在上一章实验“14_noblockio”的基础上完成，在其中加入异步通知相关内容即可，当按键按下以后驱动程序向应用程序发送 `SIGIO` 信号，应用程序获取到 `SIGIO` 信号以后读取并且打印出按键值。

17.3.1 修改设备树文件

因为是在实验“14_noblockio”的基础上完成的，因此不需要修改设备树。

17.3.2 程序编写

新建名为“15_asyncnoti”的文件夹，然后在 15_asyncnoti 文件夹里面创建 vscode 工程，工作区命名为“asyncnoti”。将“14_noblockio”实验中的 noblockio.c 复制到 15_asyncnoti 文件夹中，并重命名为 asyncnoti.c。接下来我们就修改 asyncnoti.c 这个文件，在其中添加异步通知的代码，完成以后的 asyncnoti.c 内容如下所示(因为是在上一章实验的 noblockio.c 文件的基础上修改的，为了减少篇幅，下面的代码有省略)：

示例代码 17.3.2.1 asyncnoti.c 文件代码段

```

32 #include <linux/fcntl.h>
33
34 #define KEY_CNT      1          /* 设备号个数 */
35 #define KEY_NAME     "key"     /* 名字 */
36
37 /* 定义按键状态 */
38 enum key_status {
39     KEY_PRESS = 0,           /* 按键按下 */
40     KEY_RELEASE,          /* 按键松开 */
41     KEY_KEEP,             /* 按键状态保持 */
42 };
43
44 /* key 设备结构体 */
45 struct key_dev{
46     dev_t devid;           /* 设备号 */
47     struct cdev cdev;      /* cdev */
48     struct class *class;   /* 类 */
49     struct device *device; /* 设备 */
50     struct device_node *nd; /* 设备节点 */
51     int key_gpio;          /* key 所使用的 GPIO 编号 */
52     struct timer_list timer; /* 按键值 */
53     int irq_num;           /* 中断号 */
54     atomic_t status;       /* 按键状态 */
55     wait_queue_head_t r_wait; /* 读等待队列头 */
56     struct fasync_struct *async_queue; /* fasync_struct 结构体 */
57 };
.....
153 static void key_timer_function(struct timer_list *arg)
154 {
155     static int last_val = 1;
156     int current_val;
157

```

```

158     /* 读取按键值并判断按键当前状态 */
159     current_val = gpio_get_value(key.key_gpio);
160     if (0 == current_val && last_val){
161         atomic_set(&key.status, KEY_PRESS);           /* 按下 */
162         wake_up_interruptible(&key.r_wait);          /* 唤醒 */
163         if(key.async_queue)
164             kill_fasync(&key.async_queue, SIGIO, POLL_IN);
165     }
166     else if (1 == current_val && !last_val) {
167         atomic_set(&key.status, KEY_RELEASE);        /* 松开 */
168         wake_up_interruptible(&key.r_wait);          /* 唤醒 */
169         if(key.async_queue)
170             kill_fasync(&key.async_queue, SIGIO, POLL_IN);
171     }
172     else
173         atomic_set(&key.status, KEY_KEEP);           /* 状态保持 */
174
175     last_val = current_val;
176 }
.....
221 /*
222  * @description   : fasync 函数, 用于处理异步通知
223  * @param - fd    : 文件描述符
224  * @param - filp  : 要打开的设备文件(文件描述符)
225  * @param - on    : 模式
226  * @return        : 负数表示函数执行失败
227  */
228 static int key_fasync(int fd, struct file *filp, int on)
229 {
230     return fasync_helper(fd, filp, on, &key.async_queue);
231 }
.....
246 /*
247  * @description   : 关闭/释放设备
248  * @param - filp  : 要关闭的设备文件(文件描述符)
249  * @return        : 0 成功;其他 失败
250  */
251 static int key_release(struct inode *inode, struct file *filp)
252 {
253     return key_fasync(-1, filp, 0);
254 }
.....
274 /* 设备操作函数 */
    
```

```

275 static struct file_operations key_fops = {
276     .owner = THIS_MODULE,
277     .open = key_open,
278     .read = key_read,
279     .write = key_write,
280     .release = key_release,
281     .poll = key_poll,
282     .fasync = key_fasync,
283 };
.....
373 module_init(mykey_init);
374 module_exit(mykey_exit);
375 MODULE_LICENSE("GPL");
376 MODULE_AUTHOR("ALIENTEK");
377 MODULE_INFO(intree, "Y");
    
```

第 32 行, 添加 `fcntl.h` 头文件, 因为要用到相关的 API 函数。

第 56 行, 在设备结构体 `key_dev` 中添加 `fasync_struct` 指针变量。

第 153~176 行, 在 `key_timer_function` 函数中, 当按键按下或松开动作发生时调用 `kill_fasync` 函数向应用程序发送 `SIGIO` 信号, 通知应用程序按键数据可以进行读取了。

第 228~231 行, 设备操作函数集 `file_operations` 结构体中的 `fasync` 函数 `key_fasync`, 该函数中直接调用 `fasync_helper` 函数进行相关处理。

第 253 行, 在 `key_release` 函数中也调用 `key_fasync` 函数释放 `fasync_struct` 指针变量。

第 282 行, 将 `key_fasync` 函数绑定到 `key_fops` 变量的 `fasync` 函数指针中。

17.3.3 编写测试 APP

测试 APP 要实现的内容很简单, 设置 `SIGIO` 信号的处理函数为 `sigio_signal_func`, 当驱动程序向应用程序发送 `SIGIO` 信号以后 `sigio_signal_func` 函数就会执行。`sigio_signal_func` 函数内容很简单, 就是通过 `read` 函数读取按键值。新建名为 `asynctotiApp.c` 的文件, 然后输入如下所示内容:

示例代码 17.3.3.2 `asynctotiApp.c` 文件代码段

```

13 #include <stdio.h>
14 #include <unistd.h>
15 #include <sys/types.h>
16 #include <sys/stat.h>
17 #include <fcntl.h>
18 #include <stdlib.h>
19 #include <string.h>
20 #include <signal.h>
21
22 static int fd;
23
24 /*
25  * SIGIO 信号处理函数
    
```

```

26 * @param - signum    : 信号值
27 * @return           : 无
28 */
29 static void sigio_signal_func(int signum)
30 {
31     unsigned int key_val = 0;
32
33     read(fd, &key_val, sizeof(unsigned int));
34     if (0 == key_val)
35         printf("Key Press\n");
36     else if (1 == key_val)
37         printf("Key Release\n");
38 }
39
40
41 /*
42 * @description      : main 主程序
43 * @param - argc     : argv 数组元素个数
44 * @param - argv     : 具体参数
45 * @return           : 0 成功;其他 失败
46 */
47 int main(int argc, char *argv[])
48 {
49     int flags = 0;
50
51     /* 判断传参个数是否正确 */
52     if(2 != argc) {
53         printf("Usage:\n"
54             "\t./asyncKeyApp /dev/key\n"
55             );
56         return -1;
57     }
58
59     /* 打开设备 */
60     fd = open(argv[1], O_RDONLY | O_NONBLOCK);
61     if(0 > fd) {
62         printf("ERROR: %s file open failed!\n", argv[1]);
63         return -1;
64     }
65
66     /* 设置信号 SIGIO 的处理函数 */
67     signal(SIGIO, sigio_signal_func);
68     fcntl(fd, F_SETOWN, getpid()); /* 将当前进程的进程号告诉给内核 */

```

```

69     flags = fcntl(fd, F_GETFD);           /*获取当前的进程状态 */
70     fcntl(fd, F_SETFL, flags | FASYNC); /* 设置进程启用异步通知功能 */
71
72
73     /* 循环轮询读取按键数据 */
74     for ( ; ; ) {
75
76         sleep(2);
77     }
78
79     /* 关闭设备 */
80     close(fd);
81     return 0;
82 }
    
```

第 29~38 行, sigio_signal_func 函数, SIGIO 信号的处理函数, 当驱动程序有效按键按下以后就会发送 SIGIO 信号, 此函数就会执行。此函数通过 read 函数读取按键状态数据, 然后通过 printf 函数打印在终端上。

第 67 行, 通过 signal 函数设置 SIGIO 信号的处理函数为 sigio_signal_func。

第 68~70 行, 设置当前进程的状态, 开启异步通知的功能。

第 74~77 行, for 循环, 等待信号产生。

17.4 运行测试

17.4.1 编译驱动程序和测试 APP

1、编译驱动程序

编写 Makefile 文件, 本章实验的 Makefile 文件和第五章实验基本一样, 只是将 obj-m 变量的值改为 asyncnoti.o, Makefile 内容如下所示:

示例代码 17.4.1.1 Makefile 文件

```

1  KERNELDIR := /home/alientek/rk3568_linux_sdk/kernel
.....
4  obj-m := asyncnoti.o
.....
11 clean:
12 $(MAKE) -C $(KERNELDIR) M=$(CURRENT_PATH) clean
    
```

第 4 行, 设置 obj-m 变量的值为 asyncnoti.o。

输入如下命令编译出驱动模块文件:

```
make ARCH=arm64 //ARCH=arm64 必须指定, 否则编译会失败
```

编译成功以后就会生成一个名为“asyncnoti.ko”的驱动模块文件。

2、编译测试 APP

输入如下命令编译测试 asyncnotiApp.c 这个测试程序:

```
/opt/atk-dlrk356x-toolchain/bin/aarch64-buildroot-linux-gnu-gcc asyncnotiApp.c -o
asyncnotiApp
```

编译成功以后就会生成 `asynctotiApp` 这个应用程序。

17.4.2 运行测试

在 Ubuntu 中将上一小节编译出来的 `blockio.ko` 和 `blockioApp` 这两个文件通过 `adb` 命令发送到开发板的 `/lib/modules/4.19.232` 目录下, 命令如下:

```
adb push asynctoti.ko asynctotiApp /lib/modules/4.19.232
```

发送成功以后进入到开发板目录 `lib/modules/4.19.232` 中, 输入如下命令加载 `asynctoti.ko` 驱动模块:

```
depmod //第一次加载驱动的时候需要运行此命令
modprobe asynctoti //加载驱动
```

驱动加载成功以后使用如下命令来测试中断:

```
./asynctotiApp /dev/key
```

使用杜邦线将图 13.2.1 中 `GPIO3_C5` 这个 IO 接到开发板的 3.3V 电压上, 模拟按键被按下, 如图 17.4.2.1 所示:

```
root@ATK-DLRK356X:/lib/modules/4.19.232# ./asynctotiApp /dev/key
Key Press
Key Release
Key Press
Key Release
Key Press
Key Release
Key Press
Key Release
```

图 17.4.2.1 读取到的按键值

从图 17.4.2.1 可以看出, 捕获到 `SIGIO` 信号, 并且按键值获取成功, 大家可以自行以后台模式运行 `asynctotiApp`, 查看一下这个应用程序的 CPU 使用率。如果要卸载驱动的话输入如下命令即可:

```
rmmmod asynctoti.ko
```

第十八章 platform 设备驱动实验

我们在前面几章编写的设备驱动都非常的简单，都是对 GPIO 进行最简单的读写操作。像 I2C、SPI、LCD 等这些复杂外设的驱动就不能这么去写了，Linux 系统要考虑到驱动的可重用性，因此提出了驱动的分离与分层这样的软件思路，在这个思路下诞生了我们将来最常打交道的 platform 设备驱动，也叫做平台设备驱动。本章我们就来学习一下 Linux 下的驱动分离与分层，以及 platform 框架下的设备驱动该如何编写。

18.1 Linux 驱动的分隔与分层

18.1.1 驱动的分隔与分离

对于 Linux 这样一个成熟、庞大、复杂的操作系统，代码的重用性非常重要，否则的话就会在 Linux 内核中存在大量无意义的重复代码。尤其是驱动程序，因为驱动程序占用了 Linux 内核代码量的大头，如果不对驱动程序加以管理，任由重复的代码肆意增加，那么用不了多久 Linux 内核的文件数量就庞大到无法接受的地步。

假如现在有三个平台 A、B 和 C，这三个平台(这里的平台说的是 SOC)上都有 MPU6050 这个 I2C 接口的六轴传感器，按照我们写裸机 I2C 驱动的时候的思路，每个平台都有一个 MPU6050 的驱动，因此编写出来的最简单的驱动框架如图 18.1.1.1 所示：

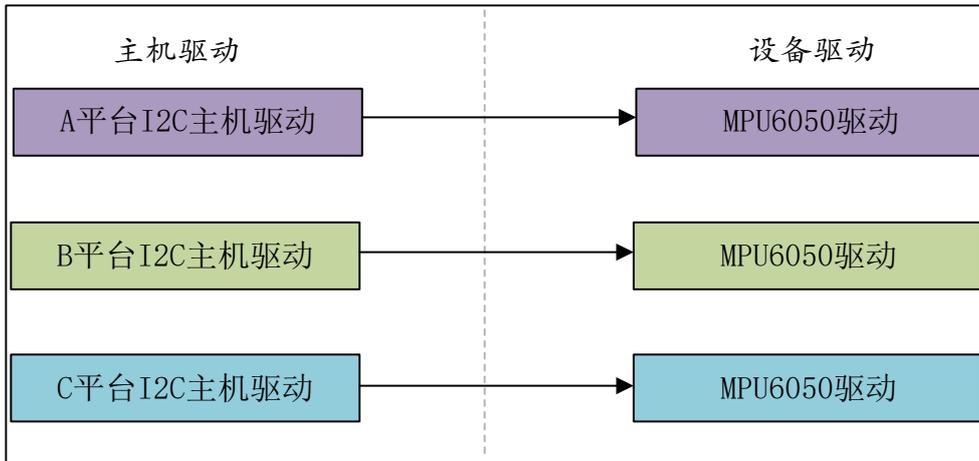


图 18.1.1.1 传统的 I2C 设备驱动

从图 18.1.1.1 可以看出，每种平台下都有一个主机驱动和设备驱动，主机驱动肯定是必须的，毕竟不同的平台其 I2C 控制器不同。但是右侧的设备驱动就没必要每个平台都写一个，因为不管对于那个 SOC 来说，MPU6050 都是一样，通过 I2C 接口读写数据就行了，只需要一个 MPU6050 的驱动程序即可。如果再来几个 I2C 设备，比如 AT24C02、FT5206(电容触摸屏)等，如果按照图 18.1.1.1 中的写法，那么设备端的驱动将会重复的编写好几次。显然在 Linux 驱动程序中这种写法是不推荐的，最好的做法就是每个平台的 I2C 控制器都提供一个统一的接口(也叫做主机驱动)，每个设备的话也只提供一个驱动程序(设备驱动)，每个设备通过统一的 I2C 接口驱动来访问，这样就可以大大简化驱动文件，比如 18.1.1.1 中三种平台下的 MPU6050 驱动框架就可以简化为图 18.1.1.2 所示：

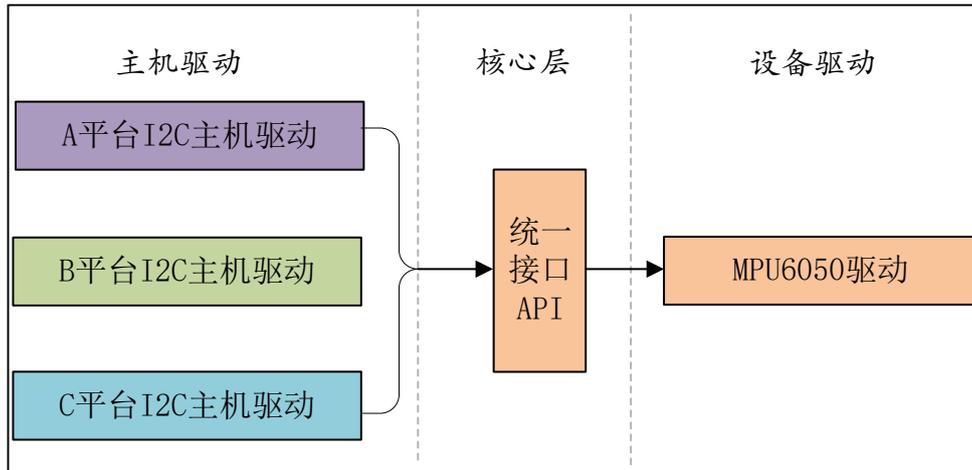


图 18.1.1.2 改进后的设备驱动

实际的 I2C 驱动设备肯定有很多种，不止 MPU6050 这一个，那么实际的驱动架构如图 18.1.1.3 所示：

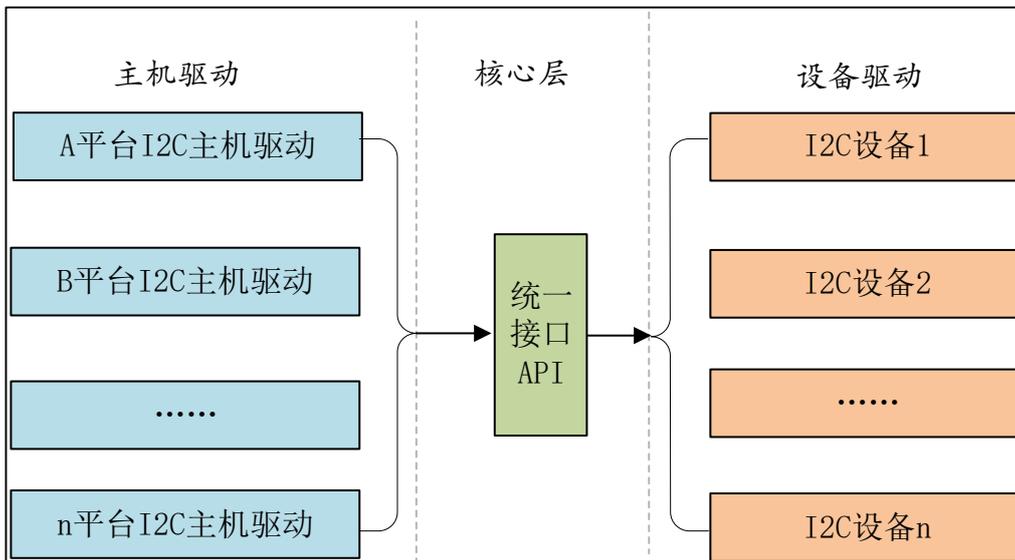


图 18.1.1.3 分隔后的驱动框架

这个就是驱动的分隔，也就是将主机驱动和设备驱动分隔开来，比如 I2C、SPI 等等都会采用驱动分隔的方式来简化驱动的开发。在实际的驱动开发中，一般 I2C 主机控制器驱动已经由半导体厂家编写好了，而设备驱动一般也由设备器件的厂家编写好了，我们只需要提供设备信息即可，比如 I2C 设备的话提供设备连接到了哪个 I2C 接口上，I2C 的速度是多少等等。相当于将设备信息从设备驱动中剥离开来，驱动使用标准方法去获取到设备信息(比如从设备树中获取到设备信息)，然后根据获取到的设备信息来初始化设备。这样就相当于驱动只负责驱动，设备只负责设备，想办法将两者进行匹配即可。这个就是 Linux 中的总线(bus)、驱动(driver)和设备(device)模型，也就是常说的驱动分离。总线就是驱动和设备信息的月老，负责给两者牵线搭桥，如图 18.1.1.4 所示：

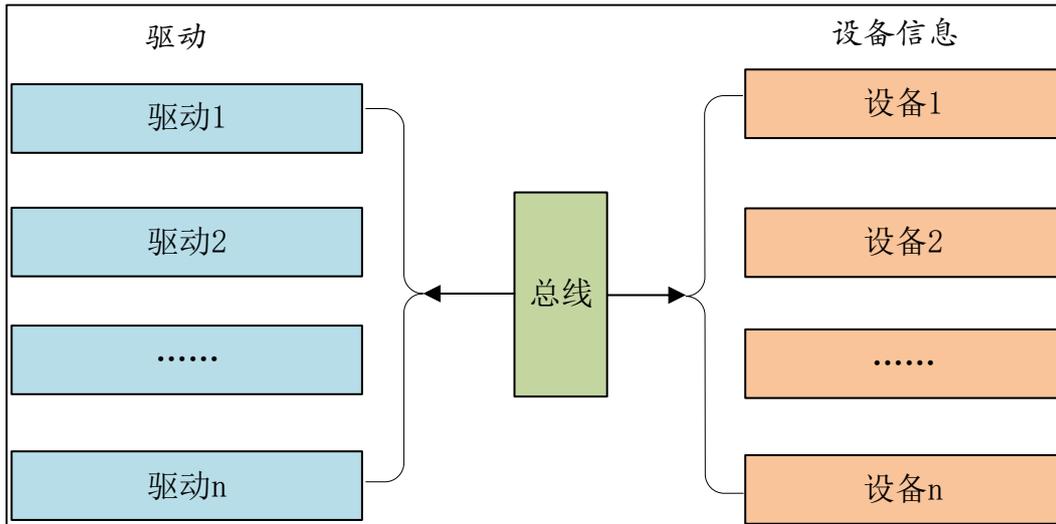


图 18.1.1.4 Linux 总线、驱动和设备模式

当我们向系统注册一个驱动的时候，总线就会在右侧的设备中查找，看看有没有与之匹配的设备，如果有的话就将两者联系起来。同样的，当向系统中注册一个设备的时候，总线就会在左侧的驱动中查找看有没有与之匹配的设备，有的话也联系起来。Linux 内核中大量的驱动程序都采用总线、驱动和设备模式，我们一会要重点讲解的 platform 驱动就是这一思想下的产物。

18.1.2 驱动的分层

上一小节讲了驱动的分隔与分离，本节我们来简单看一下驱动的分层，大家应该听说过网络的 7 层模型，不同的层负责不同的内容。同样的，Linux 下的驱动往往也是分层的，分层的目的是为了在不同的层处理不同的内容。以其他书籍或者资料常常使用到的 input(输入子系统，后面会有专门的章节详细的讲解)为例，简单介绍一下驱动的分层。input 子系统负责管理所有跟输入有关的驱动，包括键盘、鼠标、触摸等，最底层的就是设备原始驱动，负责获取输入设备的原始值，获取到的输入事件上报给 input 核心层。input 核心层会处理各种 IO 模型，并且提供 file_operations 操作集合。我们在编写输入设备驱动的时候只需要处理好输入事件的上报即可，至于如何处理这些上报的输入事件那是上层去考虑的，我们不用管。可以看出借助分层模型可以极大的简化我们的驱动编写，对于驱动编写来说非常的友好。

18.2 platform 平台驱动模型简介

前面我们讲了设备驱动的分层，并且引出了总线(bus)、驱动(driver)和设备(device)模型，比如 I2C、SPI、USB 等总线。在 SOC 中有些外设是没有总线这个概念的，但是又要使用总线、驱动和设备模型该怎么办呢？为了解决此问题，Linux 提出了 platform 这个虚拟总线，相应的就有 platform_driver 和 platform_device。

18.2.1 platform 总线

Linux 系统内核使用 bus_type 结构体表示总线，此结构体定义在文件 include/linux/device.h，bus_type 结构体内容如下：

示例代码 18.2.1.1 bus_type 结构体代码段

```
1 struct bus_type {
```

```

2  const char    1 *name;
3  const char    *dev_name;
4  struct device *dev_root;
5  const struct attribute_group **bus_groups;
6  const struct attribute_group **dev_groups;
7  const struct attribute_group **drv_groups;
8  int (*match)(struct device *dev, struct device_driver *drv);
9  int (*uevent)(struct device *dev, struct kobj_uevent_env *env);
10 int (*probe)(struct device *dev);
11 int (*remove)(struct device *dev);
12 void (*shutdown)(struct device *dev);
13 int (*online)(struct device *dev);
14 int (*offline)(struct device *dev);
15 int (*suspend)(struct device *dev, pm_message_t state);
16 int (*resume)(struct device *dev);
17 int (*num_vf)(struct device *dev);
18 int (*dma_configure)(struct device *dev);
19 const struct dev_pm_ops *pm;
20 const struct iommu_ops *iommu_ops;
21 struct subsys_private *p;
22 struct lock_class_key lock_key;
23 bool need_parent_lock;
24 };
    
```

第 8 行, match 函数, 此函数很重要, 单词 match 的意思就是“匹配、相配”, 因此此函数就是完成设备和驱动之间匹配的, 总线就是使用 match 函数来根据注册的设备来查找对应的驱动, 或者根据注册的驱动来查找相应的设备, 因此每一条总线都必须实现此函数。match 函数有两个参数: dev 和 drv, 这两个参数分别为 device 和 device_driver 类型, 也就是设备 和驱动。

platform 总线是 bus_type 的一个具体实例, 定义在文件 drivers/base/platform.c, platform 总线定义如下:

示例代码 18.2.1.2 platform 总线实例

```

1  struct bus_type platform_bus_type = {
2      .name          = "platform",
3      .dev_groups    = platform_dev_groups,
4      .match         = platform_match,
5      .uevent        = platform_uevent,
6      .dma_configure = platform_dma_configure,
7      .pm            = &platform_dev_pm_ops,
8  };
    
```

platform_bus_type 就是 platform 平台总线, 其中 platform_match 就是匹配函数。我们来看一下驱动和设备是如何匹配的, platform_match 函数定义在文件 drivers/base/platform.c 中, 函数内容如下所示:

示例代码 18.2.1.3 platform 总线实例

```

1  static int platform_match(struct device *dev,
    
```

```

                struct device_driver *drv)
2  {
3      struct platform_device *pdev = to_platform_device(dev);
4      struct platform_driver *pdrv = to_platform_driver(drv);
5
6      /*When driver_override is set, only bind to the matching driver*/
7      if (pdev->driver_override)
8          return !strcmp(pdev->driver_override, drv->name);
9
10     /* Attempt an OF style match first */
11     if (of_driver_match_device(dev, drv))
12         return 1;
13
14     /* Then try ACPI style match */
15     if (acpi_driver_match_device(dev, drv))
16         return 1;
17
18     /* Then try to match against the id table */
19     if (pdrv->id_table)
20         return platform_match_id(pdrv->id_table, pdev) != NULL;
21
22     /* fall-back to driver name match */
23     return (strcmp(pdev->name, drv->name) == 0);
24 }
    
```

驱动和设备的匹配有四种方法，我们依次来看一下：

第 11~12 行，第一种匹配方式，OF 类型的匹配，也就是设备树采用的匹配方式，`of_driver_match_device` 函数定义在文件 `include/linux/of_device.h` 中。`device_driver` 结构体(表示设备驱动)中有个名为 `of_match_table` 的成员变量，此成员变量保存着驱动的 `compatible` 匹配表，设备树中的每个设备节点的 `compatible` 属性会和 `of_match_table` 表中的所有成员比较，查看是否有相同的条目，如果有的话就表示设备和此驱动匹配，设备和驱动匹配成功以后 `probe` 函数就会执行。

第 15~16 行，第二种匹配方式，ACPI 匹配方式。

第 19~20 行，第三种匹配方式，`id_table` 匹配，每个 `platform_driver` 结构体有一个 `id_table` 成员变量，顾名思义，保存了很多 `id` 信息。这些 `id` 信息存放着这个 `platformd` 驱动所支持的驱动类型。

第 23 行，第四种匹配方式，如果第三种匹配方式的 `id_table` 不存在的话就直接比较驱动和设备的 `name` 字段，看看是不是相等，如果相等的话就匹配成功。

对于支持设备树的 Linux 版本号，一般设备驱动为了兼容性都支持设备树和无设备树两种匹配方式。也就是第一种匹配方式一般都会存在，第三种和第四种只要存在一种就可以，一般用的最多的还是第四种，也就是直接比较驱动和设备的 `name` 字段，毕竟这种方式最简单了。

18.2.2 platform 驱动

`platform_driver` 结构体表示 `platform` 驱动，此结构体定义在文件

include/linux/platform_device.h 中, 内容如下:

示例代码 18.2.2.1 platform_driver 结构体

```

1 struct platform_driver {
2     int (*probe)(struct platform_device *);
3     int (*remove)(struct platform_device *);
4     void (*shutdown)(struct platform_device *);
5     int (*suspend)(struct platform_device *, pm_message_t state);
6     int (*resume)(struct platform_device *);
7     struct device_driver driver;
8     const struct platform_device_id *id_table;
9     bool prevent_deferred_probe;
10 };
    
```

第 2 行, probe 函数, 当驱动与设备匹配成功以后 probe 函数就会执行, 非常重要的函数!! 一般驱动的提供者会编写, 如果自己要编写一个全新的驱动, 那么 probe 就需要自行实现。

第 7 行, driver 成员, 为 device_driver 结构体变量, Linux 内核里面大量使用到了面向对象的思维, device_driver 相当于基类, 提供了最基础的驱动框架。platform_driver 继承了这个基类, 然后在此基础上又添加了一些特有的成员变量。

第 8 行, id_table 表, 也就是我们上一小节讲解 platform 总线匹配驱动和设备的时候采用的第三种方法, id_table 是个表(也就是数组), 每个元素的类型为 platform_device_id, platform_device_id 结构体内容如下:

示例代码 18.2.2.2 platform_device_id 结构体

```

1 struct platform_device_id {
2     char name[PLATFORM_NAME_SIZE];
3     kernel_ulong_t driver_data;
4 };
    
```

device_driver 结构体定义在 include/linux/device.h, device_driver 结构体内容如下:

示例代码 18.2.2.3 device_driver 结构体

```

1 struct device_driver {
2     const char *name;
3     struct bus_type *bus;
4     struct module *owner;
5     const char *mod_name; /* used for built-in modules */
6     bool suppress_bind_attrs; /* disables bind/unbind via sysfs */
7     enum probe_type probe_type;
8     const struct of_device_id *of_match_table;
9     const struct acpi_device_id *acpi_match_table;
10    int (*probe)(struct device *dev);
11    int (*remove)(struct device *dev);
12    void (*shutdown)(struct device *dev);
13    int (*suspend)(struct device *dev, pm_message_t state);
14    int (*resume)(struct device *dev);
15    const struct attribute_group **groups;
16    const struct dev_pm_ops *pm;
    
```

```

17 void (*coredump) (struct device *dev);
18 struct driver_private *p;
19 };
    
```

第 8 行, `of_match_table` 就是采用设备树的时候驱动使用的匹配表, 同样是数组, 每个匹配项都为 `of_device_id` 结构体类型, 此结构体定义在文件 `include/linux/mod_devicetable.h` 中, 内容如下:

示例代码 18.2.2.4 `of_device_id` 结构体

```

1 struct of_device_id {
2     char          name[32];
3     char          type[32];
4     char          compatible[128];
5     const void    *data;
6 };
    
```

第 4 行的 `compatible` 非常重要, 因为对于设备树而言, 就是通过设备节点的 `compatible` 属性值和 `of_match_table` 中每个项目的 `compatible` 成员变量进行比较, 如果有相等的就表示设备和此驱动匹配成功。

在编写 `platform` 驱动的时候, 首先定义一个 `platform_driver` 结构体变量, 然后实现结构体中的各个成员变量, 重点是实现匹配方法以及 `probe` 函数。当驱动和设备匹配成功以后 `probe` 函数就会执行, 具体的驱动程序在 `probe` 函数里面编写, 比如字符设备驱动等等。

当我们定义并初始化好 `platform_driver` 结构体变量以后, 需要在驱动入口函数里面调用 `platform_driver_register` 函数向 Linux 内核注册一个 `platform` 驱动, `platform_driver_register` 函数原型如下所示:

```
int platform_driver_register(struct platform_driver *driver)
```

函数参数和返回值含义如下:

driver: 要注册的 `platform` 驱动。

返回值: 负数, 失败; 0, 成功。

还需要在驱动卸载函数中通过 `platform_driver_unregister` 函数卸载 `platform` 驱动, `platform_driver_unregister` 函数原型如下:

```
void platform_driver_unregister(struct platform_driver *drv)
```

函数参数和返回值含义如下:

drv: 要卸载的 `platform` 驱动。

返回值: 无。

`platform` 驱动框架如下所示:

示例代码 18.2.2.5 `platform` 驱动框架

```

/* 设备结构体 */
1 struct xxx_dev{
2     struct cdev cdev;
3     /* 设备结构体其他具体内容 */
4 };
5
6 struct xxx_dev xxxdev; /* 定义个设备结构体变量 */
7
8 static int xxx_open(struct inode *inode, struct file *filp)
    
```

```

9  {
10     /* 函数具体内容 */
11     return 0;
12 }
13
14 static ssize_t xxx_write(struct file *filp, const char __user *buf,
15                          size_t cnt, loff_t *offt)
16 {
17     /* 函数具体内容 */
18     return 0;
19 }
20 /*
21  * 字符设备驱动操作集
22  */
23 static struct file_operations xxx_fops = {
24     .owner = THIS_MODULE,
25     .open = xxx_open,
26     .write = xxx_write,
27 };
28
29 /*
30  * platform 驱动的 probe 函数
31  * 驱动与设备匹配成功以后此函数就会执行
32  */
33 static int xxx_probe(struct platform_device *dev)
34 {
35     .....
36     cdev_init(&xxxdev.cdev, &xxx_fops); /* 注册字符设备驱动 */
37     /* 函数具体内容 */
38     return 0;
39 }
40
41 static int xxx_remove(struct platform_device *dev)
42 {
43     .....
44     cdev_del(&xxxdev.cdev); /* 删除 cdev */
45     /* 函数具体内容 */
46     return 0;
47 }
48
49 /* 匹配列表 */
50 static const struct of_device_id xxx_of_match[] = {

```

```

51     { .compatible = "xxx-gpio" },
52     { /* Sentinel */ }
53 };
54
55 /*
56 * platform 平台驱动结构体
57 */
58 static struct platform_driver xxx_driver = {
59     .driver = {
60         .name          = "xxx",
61         .of_match_table = xxx_of_match,
62     },
63     .probe             = xxx_probe,
64     .remove            = xxx_remove,
65 };
66
67 /* 驱动模块加载 */
68 static int __init xxxdriver_init(void)
69 {
70     return platform_driver_register(&xxx_driver);
71 }
72
73 /* 驱动模块卸载 */
74 static void __exit xxxdriver_exit(void)
75 {
76     platform_driver_unregister(&xxx_driver);
77 }
78
79 module_init(xxxdriver_init);
80 module_exit(xxxdriver_exit);
81 MODULE_LICENSE("GPL");
82 MODULE_AUTHOR("alientek");
    
```

第 1~27 行, 传统的字符设备驱动, 所谓的 platform 驱动并不是独立于字符设备驱动、块设备驱动和网络设备驱动之外的其他种类的驱动。platform 只是为了驱动的分离与分层而提出的一种框架, 其驱动的具体实现还是需要字符设备驱动、块设备驱动或网络设备驱动。

第 33~39 行, xxx_probe 函数, 当驱动和设备匹配成功以后此函数就会执行, 以前在驱动入口 init 函数里面编写的字符设备驱动程序就全部放到此 probe 函数里面。比如注册字符设备驱动、添加 cdev、创建类等等。

第 41~47 行, xxx_remove 函数, platform_driver 结构体中的 remove 成员变量, 当关闭 platform 设备驱动的时候此函数就会执行, 以前在驱动卸载 exit 函数里面要做的事情就放到此函数中来。比如, 使用 iounmap 释放内存、删除 cdev, 注销设备号等等。

第 50~53 行, xxx_of_match 匹配表, 如果使用设备树的话将通过此匹配表进行驱动和设备的匹配。第 51 行设置了一个匹配项, 此匹配项的 compatible 值为“xxx-gpio”, 因此当设备树中

设备节点的 `compatible` 属性值为“xxx-gpio”的时候此设备就会与此驱动匹配。第 52 行是一个标记, `of_device_id` 表最后一个匹配项必须是空的。

第 58~65 行, 定义一个 `platform_driver` 结构体变量 `xxx_driver`, 表示 `platform` 驱动, 第 59~62 行设置 `platform_driver` 中的 `device_driver` 成员变量的 `name` 和 `of_match_table` 这两个属性。其中 `name` 属性用于传统的驱动与设备匹配, 也就是检查驱动和设备的 `name` 字段是不是相同。`of_match_table` 属性就是用于设备树下的驱动与设备检查。对于一个完整的驱动程序, 必须提供有设备树和无设备树两种匹配方法。最后 63 和 64 这两行设置 `probe` 和 `remove` 这两成员变量。

第 68~71 行, 驱动入口函数, 调用 `platform_driver_register` 函数向 Linux 内核注册一个 `platform` 驱动, 也就是上面定义的 `xxx_driver` 结构体变量。

第 74~77 行, 驱动出口函数, 调用 `platform_driver_unregister` 函数卸载前面注册的 `platform` 驱动。

总体来说, `platform` 驱动还是传统的字符设备驱动、块设备驱动或网络设备驱动, 只是套上了一张“`platform`”的皮, 目的是为了使用总线、驱动和设备这个驱动模型来实现驱动的分离与分层。

18.2.3 platform 设备

`platform` 驱动已经准备好了, 我们还需要 `platform` 设备, 否则的话单单一个驱动也做不了什么。`platform_device` 这个结构体表示 `platform` 设备, 这里我们要注意, 如果内核支持设备树的话就不要再使用 `platform_device` 来描述设备了, 因为改用设备树去描述了。当然了, 你一定要用 `platform_device` 来描述设备信息的话也是可以的。`platform_device` 结构体定义在文件 `include/linux/platform_device.h` 中, 结构体内容如下:

示例代码 18.2.3.1 platform_device 结构体代码段

```
1 struct platform_device {
2     const char *name;
3     int id;
4     bool id_auto;
5     struct device dev;
6     u32 num_resources;
7     struct resource *resource;
8
9     const struct platform_device_id *id_entry;
10    char *driver_override; /* Driver name to force a match */
11
12    /* MFD cell pointer */
13    struct mfd_cell *mfd_cell;
14
15    /* arch specific additions */
16    struct pdev_archdata archdata;
19 };
```

第 2 行, `name` 表示设备名字, 要和所使用的 `platform` 驱动的 `name` 字段相同, 否则的话设备就无法匹配到对应的驱动。比如对应的 `platform` 驱动的 `name` 字段为“xxx-gpio”, 那么此 `name` 字段也要设置为“xxx-gpio”。

第 6 行, `num_resources` 表示资源数量, 一般为第 7 行 `resource` 资源的大小。

第 7 行, resource 表示资源, 也就是设备信息, 比如外设寄存器等。Linux 内核使用 resource 结构体表示资源, resource 结构体定义在 include/linux/ioport.h 文件里面, 内容为:

示例代码 18.2.3.2 resource 结构体代码段

```
1 struct resource {
2     resource_size_t start;
3     resource_size_t end;
4     const char *name;
5     unsigned long flags;
6     unsigned long desc;
7     struct resource *parent, *sibling, *child;
8 };
```

start 和 end 分别表示资源的起始和终止信息, 对于内存类的资源, 就表示内存起始和终止地址, name 表示资源名字, flags 表示资源类型, 可选的资源类型都定义在了文件 include/linux/ioport.h 里面, 如下所示:

示例代码 18.2.3.3 资源类型

```
1 #define IORESOURCE_BITS          0x000000ff /* Bus-specific bits */
2
3 #define IORESOURCE_TYPE_BITS     0x00001f00 /* Resource type */
4 #define IORESOURCE_IO           0x00000100 /* 表示 IO 口的资源 */
5 #define IORESOURCE_MEM          0x00000200 /* 表示内存地址 */
6 #define IORESOURCE_REG          0x00000300 /* Register offsets */
7 #define IORESOURCE_IRQ          0x00000400 /* 中断号 */
8 #define IORESOURCE_DMA          0x00000800 /* DMA 通道号 */
9 #define IORESOURCE_BUS          0x00001000 /* 总线号 */
10 .....
84 #define IORESOURCE_PCI_EA_BEI (1<<5) /* BAR Equivalent Indicator */
```

在以前不支持设备树的 Linux 版本中, 用户需要编写 platform_device 变量来描述设备信息, 然后使用 platform_device_register 函数将设备信息注册到 Linux 内核中, 此函数原型如下所示:

```
int platform_device_register(struct platform_device *pdev)
```

函数参数和返回值含义如下:

pdev: 要注册的 platform 设备。

返回值: 负数, 失败; 0, 成功。

如果不再使用 platform 的话可以通过 platform_device_unregister 函数注销掉相应的 platform 设备, platform_device_unregister 函数原型如下:

```
void platform_device_unregister(struct platform_device *pdev)
```

函数参数和返回值含义如下:

pdev: 要注销的 platform 设备。

返回值: 无。

platform 设备信息框架如下所示:

示例代码 18.2.3.4 platform 设备框架

```
1 /* 寄存器地址定义 */
2 #define PERIPH1_REGISTER_BASE    (0X20000000) /* 外设 1 寄存器首地址 */
3 #define PERIPH2_REGISTER_BASE    (0X020E0068) /* 外设 2 寄存器首地址 */
```

```

4 #define REGISTER_LENGTH          4
5
6 /* 资源 */
7 static struct resource xxx_resources[] = {
8     [0] = {
9         .start = PERIPH1_REGISTER_BASE,
10        .end   = (PERIPH1_REGISTER_BASE + REGISTER_LENGTH - 1),
11        .flags = IORESOURCE_MEM,
12    },
13    [1] = {
14        .start = PERIPH2_REGISTER_BASE,
15        .end   = (PERIPH2_REGISTER_BASE + REGISTER_LENGTH - 1),
16        .flags = IORESOURCE_MEM,
17    },
18 };
19
20 /* platform 设备结构体 */
21 static struct platform_device xxxdevice = {
22     .name = "xxx-gpio",
23     .id = -1,
24     .num_resources = ARRAY_SIZE(xxx_resources),
25     .resource = xxx_resources,
26 };
27
28 /* 设备模块加载 */
29 static int __init xxxdevice_init(void)
30 {
31     return platform_device_register(&xxxdevice);
32 }
33
34 /* 设备模块注销 */
35 static void __exit xxx_resourcesdevice_exit(void)
36 {
37     platform_device_unregister(&xxxdevice);
38 }
39
40 module_init(xxxdevice_init);
41 module_exit(xxxdevice_exit);
42 MODULE_LICENSE("GPL");
43 MODULE_AUTHOR("alientek");
    
```

第 7~18 行, 数组 `xxx_resources` 表示设备资源, 一共有两个资源, 分别为设备外设 1 和外设 2 的寄存器信息。因此 `flags` 都为 `IORESOURCE_MEM`, 表示资源为内存类型的。

第 21~26 行, platform 设备结构体变量, 注意 name 字段要和所使用的驱动中的 name 字段一致, 否则驱动和设备无法匹配成功。num_resources 表示资源大小, 其实就是数组 xxx_resources 的元素数量, 这里用 ARRAY_SIZE 来测量一个数组的元素个数。

第 29~32 行, 设备模块加载函数, 在此函数中调用 platform_device_register 向 Linux 内核注册 platform 设备。

第 35~38 行, 设备模块卸载函数, 在此函数中调用 platform_device_unregister 从 Linux 内核中卸载 platform 设备。

示例代码 18.2.3.4 主要是在不支持设备树的 Linux 版本中使用的, 当 Linux 内核支持了设备树以后就不需要用户手动去注册 platform 设备了。因为设备信息都放到了设备树中去描述, Linux 内核启动的时候会从设备树中读取设备信息, 然后将其组织成 platform_device 形式, 至于设备树到 platform_device 的具体过程就不去详细的追究了, 感兴趣的可以去看一下, 网上也有很多博客详细的讲解了整个过程。

关于 platform 下的总线、驱动和设备就讲解到这里, 我们接下来就使用 platform 驱动框架来编写一个 LED 灯驱动, 本章我们不使用设备树来描述设备信息, 我们采用自定义 platform_device 这种“古老”方式来编写 LED 的设备信息。下一章我们来编写设备树下的 platform 驱动, 这样我们就掌握了无设备树和有设备树这两种 platform 驱动的开发方式。

18.3 硬件原理图分析

本章实验我们只使用到正点原子的 ATK-DLRK3568 开发板上的 LED, 因此实验硬件原理图参考 6.2 小节即可。

18.4 试验程序编写

本实验对应的例程路径为: [开发板光盘](#)→01、[程序源码](#)→Linux 驱动例程→16_platform。

本章实验我们需要编写一个驱动模块和一个设备模块, 其中驱动模块是 platform 驱动程序, 设备模块是 platform 的设备信息。当这两个模块都加载成功以后就会匹配成功, 然后 platform 驱动模块中的 probe 函数就会执行, probe 函数中就是传统的字符设备驱动那一套。

18.4.1 platform 设备与驱动程序编写

新建名为“16_platform”的文件夹, 然后在 16_platform 文件夹里面创建 vscode 工程, 工作区命名为“platform”。新建名为 leddevice.c 和 leddriver.c 这两个文件, 这两个文件分别为 LED 灯的 platform 设备文件和 LED 灯的 platform 的驱动文件。在 leddevice.c 中输入如下所示内容:

示例代码 18.4.1.1 leddevice.c 文件代码段

```

1  #include <linux/types.h>
2  #include <linux/kernel.h>
3  #include <linux/delay.h>
4  #include <linux/ide.h>
5  #include <linux/init.h>
6  #include <linux/module.h>
7  #include <linux/errno.h>
8  #include <linux/gpio.h>
9  #include <linux/cdev.h>
10 #include <linux/device.h>
    
```

```

11 #include <linux/of_gpio.h>
12 #include <linux/semaphore.h>
13 #include <linux/timer.h>
14 #include <linux/irq.h>
15 #include <linux/wait.h>
16 #include <linux/poll.h>
17 #include <linux/fs.h>
18 #include <linux/fcntl.h>
19 #include <linux/platform_device.h>
20 #include <linux/fcntl.h>
21 // #include <asm/mach/map.h>
22 #include <asm/uaccess.h>
23 #include <asm/io.h>
24
25 /* 寄存器物理地址 */
26 #define PMU_GRF_BASE                (0xFDC20000)
27 #define PMU_GRF_GPIO0C_IOMUX_L      (PMU_GRF_BASE + 0x0010)
28 #define PMU_GRF_GPIO0C_DS_0        (PMU_GRF_BASE + 0x100EC)
29
30 #define GPIO0_BASE                  (0xFDD60000)
31 #define GPIO0_SWPORT_DR_H           (GPIO0_BASE + 0x0004)
32 #define GPIO0_SWPORT_DDR_H         (GPIO0_BASE + 0x000C)
33 #define REGISTER_LENGTH             4
34
35 /* @description      : 释放 flatform 设备模块的时候此函数会执行
36 * @param - dev      : 要释放的设备
37 * @return           : 无
38 */
39 static void led_release(struct device *dev)
40 {
41     printk("led device released!\r\n");
42 }
43
44 /*
45 * 设备资源信息，也就是 LED0 所使用的所有寄存器
46 */
47 static struct resource led_resources[] = {
48     [0] = {
49         .start = PMU_GRF_GPIO0C_IOMUX_L,
50         .end   = (PMU_GRF_GPIO0C_IOMUX_L + REGISTER_LENGTH - 1),
51         .flags = IORESOURCE_MEM,
52     },
53     [1] = {

```

```

54     .start = PMU_GRF_GPIO0C_DS_0,
55     .end   = (PMU_GRF_GPIO0C_DS_0 + REGISTER_LENGTH - 1),
56     .flags = IORESOURCE_MEM,
57 },
58 [2] = {
59     .start = GPIO0_SWPORT_DR_H,
60     .end   = (GPIO0_SWPORT_DR_H + REGISTER_LENGTH - 1),
61     .flags = IORESOURCE_MEM,
62 },
63 [3] = {
64     .start = GPIO0_SWPORT_DDR_H,
65     .end   = (GPIO0_SWPORT_DDR_H + REGISTER_LENGTH - 1),
66     .flags = IORESOURCE_MEM,
67 },
68 };
69
70
71 /*
72  * platform 设备结构体
73  */
74 static struct platform_device leddevice = {
75     .name = "rk3568-led",
76     .id = -1,
77     .dev = {
78         .release = &led_release,
79     },
80     .num_resources = ARRAY_SIZE(led_resources),
81     .resource = led_resources,
82 };
83
84 /*
85  * @description    : 设备模块加载
86  * @param          : 无
87  * @return         : 无
88  */
89 static int __init leddevice_init(void)
90 {
91     return platform_device_register(&leddevice);
92 }
93
94 /*
95  * @description    : 设备模块注销
96  * @param          : 无
    
```

```

97  * @return      : 无
98  */
99  static void __exit leddevice_exit(void)
100 {
101     platform_device_unregister(&leddevice);
102 }
103
104 module_init(leddevice_init);
105 module_exit(leddevice_exit);
106 MODULE_LICENSE("GPL");
107 MODULE_AUTHOR("ALIENTEK");
108 MODULE_INFO(intree, "Y");
    
```

leddevice.c 文件内容就是按照示例代码 18.2.3.4 的 platform 设备模板编写的。

第 47~68 行, led_resources 数组, 也就是设备资源, 描述了 LED 所要使用到的寄存器信息, 也就是 IORESOURCE_MEM 资源。

第 74~82, platform 设备结构体变量 leddevice, 这里要注意 name 字段为“rk3568-led”, 所以稍后编写 platform 驱动中的 name 字段也要为“rk3568-led”, 否则设备和驱动匹配失败。

第 89~92 行, 设备模块加载函数, 在此函数里面通过 platform_device_register 向 Linux 内核注册 leddevice 这个 platform 设备。

第 99~102 行, 设备模块卸载函数, 在此函数里面通过 platform_device_unregister 从 Linux 内核中删除掉 leddevice 这个 platform 设备。

leddevice.c 文件编写完成以后就编写 leddriver.c 这个 platform 驱动文件, 在 leddriver.c 里面输入如下内容:

示例代码 54.4.1.2 leddriver.c 文件代码段

```

1  #include <linux/types.h>
2  #include <linux/kernel.h>
3  #include <linux/delay.h>
4  #include <linux/ide.h>
5  #include <linux/init.h>
6  #include <linux/module.h>
7  #include <linux/errno.h>
8  #include <linux/gpio.h>
9  #include <linux/cdev.h>
10 #include <linux/device.h>
11 #include <linux/of_gpio.h>
12 #include <linux/semaphore.h>
13 #include <linux/timer.h>
14 #include <linux/irq.h>
15 #include <linux/wait.h>
16 #include <linux/poll.h>
17 #include <linux/fs.h>
18 #include <linux/fcntl.h>
19 #include <linux/platform_device.h>
    
```

```

20 // #include <asm/mach/map.h>
21 #include <asm/uaccess.h>
22 #include <asm/io.h>
23
24 #define LEDDEV_CNT          1                /* 设备号长度      */
25 #define LEDDEV_NAME        "platled"       /* 设备名字        */
26 #define LEDOFF             0
27 #define LEDON              1
28
29 /* 映射后的寄存器虚拟地址指针 */
30 static void __iomem *PMU_GRP_GPIO0C_IOMUX_L_PI;
31 static void __iomem *PMU_GRP_GPIO0C_DS_0_PI;
32 static void __iomem *GPIO0_SWPORT_DR_H_PI;
33 static void __iomem *GPIO0_SWPORT_DDR_H_PI;
34
35 /* leddev 设备结构体 */
36 struct leddev_dev{
37     dev_t devid;                /* 设备号      */
38     struct cdev cdev;          /* cdev        */
39     struct class *class;       /* 类          */
40     struct device *device;     /* 设备        */
41 };
42
43 struct leddev_dev leddev;     /* led 设备    */
44
45 /*
46  * @description      : LED 打开/关闭
47  * @param - sta      : LEDON(0) 打开 LED, LEDOFF(1) 关闭 LED
48  * @return           : 无
49  */
50 void led_switch(u8 sta)
51 {
52     u32 val = 0;
53     if(sta == LEDON) {
54         val = readl(GPIO0_SWPORT_DR_H_PI);
55         val &= ~(0X1 << 0); /* bit0 清零*/
56         val |= ((0X1 << 16) | (0X1 << 0)); /* bit16 置 1, 允许写 bit0,
57                                             bit0, 高电平*/
58         writel(val, GPIO0_SWPORT_DR_H_PI);
59     }else if(sta == LEDOFF) {
60         val = readl(GPIO0_SWPORT_DR_H_PI);
61         val &= ~(0X1 << 0); /* bit0 清零*/
62         val |= ((0X1 << 16) | (0X0 << 0)); /* bit16 置 1, 允许写 bit0,
    
```

```

63             bit0, 低电平 */
64     writel(val, GPIO0_SWPORT_DR_H_PI);
65 }
66 }
67
68 /*
69  * @description      : 取消映射
70  * @return           : 无
71  */
72 void led_unmap(void)
73 {
74     /* 取消映射 */
75     iounmap(PMU_GRF_GPIO0C_IOMUX_L_PI);
76     iounmap(PMU_GRF_GPIO0C_DS_0_PI);
77     iounmap(GPIO0_SWPORT_DR_H_PI);
78     iounmap(GPIO0_SWPORT_DDR_H_PI);
79 }
80
81 /*
82  * @description      : 打开设备
83  * @param - inode    : 传递给驱动的 inode
84  * @param - filp     : 设备文件, file 结构体有个叫做 private_data 的成员变量
85  *                   一般在 open 的时候将 private_data 指向设备结构体。
86  * @return           : 0 成功;其他 失败
87  */
88 static int led_open(struct inode *inode, struct file *filp)
89 {
90     return 0;
91 }
92
93 /*
94  * @description      : 向设备写数据
95  * @param - filp     : 设备文件, 表示打开的文件描述符
96  * @param - buf      : 要写给设备写入的数据
97  * @param - cnt      : 要写入的数据长度
98  * @param - offt     : 相对于文件首地址的偏移
99  * @return           : 写入的字节数, 如果为负值, 表示写入失败
100 */
101 static ssize_t led_write(struct file *filp, const char __user *buf,
102 size_t cnt, loff_t *offt)
103 {
104     int retvalue;
105     unsigned char databuf[1];

```

```

105     unsigned char ledstat;
106
107     retvalue = copy_from_user(databuf, buf, cnt);
108     if(retvalue < 0) {
109         printk("kernel write failed!\r\n");
110         return -EFAULT;
111     }
112
113     ledstat = databuf[0];           /* 获取状态值 */
114     if(ledstat == LEDON) {
115         led_switch(LEDON);         /* 打开 LED 灯 */
116     }else if(ledstat == LEDOFF) {
117         led_switch(LEDOFF);       /* 关闭 LED 灯 */
118     }
119
120     return 0;
121 }
122
123 /* 设备操作函数 */
124 static struct file_operations led_fops = {
125     .owner = THIS_MODULE,
126     .open = led_open,
127     .write = led_write,
128 };
129
130 /*
131  * @description      : flatform 驱动的 probe 函数
132  * @param - dev     : platform 设备
133  * @return           : 0, 成功;其他负值,失败
134  */
135 static int led_probe(struct platform_device *dev)
136 {
137     int i = 0, ret;
138     int ressize[4];
139     u32 val = 0;
140     struct resource *ledsource[4];
141
142     printk("led driver and device has matched!\r\n");
143     /* 1、获取资源 */
144     for (i = 0; i < 4; i++) {
145         ledsource[i] = platform_get_resource(dev, IORESOURCE_MEM, i);
146         if (!ledsource[i]) {
147             dev_err(&dev->dev, "No MEM resource for always on\n");

```

```

148         return -ENXIO;
149     }
150     ressize[i] = resource_size(ledsource[i]);
151 }
152
153 /* 2、初始化 LED */
154 /* 寄存器地址映射 */
155     PMU_GRF_GPIO0C_IOMUX_L_PI = ioremap(ledsource[0]->start,
ressize[0]);
156     PMU_GRF_GPIO0C_DS_0_PI = ioremap(ledsource[1]->start, ressize[1]);
157     GPIO0_SWPORT_DR_H_PI = ioremap(ledsource[2]->start, ressize[2]);
158     GPIO0_SWPORT_DDR_H_PI = ioremap(ledsource[3]->start, ressize[3]);
159
160 /* 3、设置 GPIO0_C0 为 GPIO 功能。*/
161 val = readl(PMU_GRF_GPIO0C_IOMUX_L_PI);
162 val &= ~(0x7 << 0); /* bit2:0, 清零 */
163 val |= ((0x7 << 16) | (0x0 << 0)); /* bit18:16 置 1, 允许写 bit2:0,
164                                     bit2:0: 0, 用作 GPIO0_C0 */
165 writel(val, PMU_GRF_GPIO0C_IOMUX_L_PI);
166
167 /* 4、设置 GPIO0_C0 驱动能力为 level5 */
168 val = readl(PMU_GRF_GPIO0C_DS_0_PI);
169 val &= ~(0x3f << 0); /* bit5:0 清零*/
170 val |= ((0x3f << 16) | (0x3f << 0)); /* bit21:16 置 1, 允许写
bit5:0,
171                                     bit5:0: 0, 用作 GPIO0_C0 */
172 writel(val, PMU_GRF_GPIO0C_DS_0_PI);
173
174 /* 5、设置 GPIO0_C0 为输出 */
175 val = readl(GPIO0_SWPORT_DDR_H_PI);
176 val &= ~(0x1 << 0); /* bit0 清零*/
177 val |= ((0x1 << 16) | (0x1 << 0)); /* bit16 置 1, 允许写 bit0,
178                                     bit0, 高电平 */
179 writel(val, GPIO0_SWPORT_DDR_H_PI);
180
181 /* 6、设置 GPIO0_C0 为低电平, 关闭 LED 灯。*/
182 val = readl(GPIO0_SWPORT_DR_H_PI);
183 val &= ~(0x1 << 0); /* bit0 清零*/
184 val |= ((0x1 << 16) | (0x0 << 0)); /* bit16 置 1, 允许写 bit0,
185                                     bit0, 低电平 */
186 writel(val, GPIO0_SWPORT_DR_H_PI);
187
188 /* 注册字符设备驱动 */
    
```

```

189     /* 1、申请设备号 */
190     ret = alloc_chrdev_region(&leddev.devid, 0, LEDDEV_CNT,
LEDDEV_NAME);
191     if(ret < 0)
192         goto fail_map;
193
194     /* 2、初始化 cdev */
195     leddev.cdev.owner = THIS_MODULE;
196     cdev_init(&leddev.cdev, &led_fops);
197
198     /* 3、添加一个 cdev */
199     ret = cdev_add(&leddev.cdev, leddev.devid, LEDDEV_CNT);
200     if(ret < 0)
201         goto del_unregister;
202
203     /* 4、创建类 */
204     leddev.class = class_create(THIS_MODULE, LEDDEV_NAME);
205     if (IS_ERR(leddev.class)) {
206         goto del_cdev;
207     }
208
209     /* 5、创建设备 */
210     leddev.device = device_create(leddev.class, NULL, leddev.devid,
NULL, LEDDEV_NAME);
211     if (IS_ERR(leddev.device)) {
212         goto destroy_class;
213     }
214     return 0;
215
216 destroy_class:
217     class_destroy(leddev.class);
218 del_cdev:
219     cdev_del(&leddev.cdev);
220 del_unregister:
221     unregister_chrdev_region(leddev.devid, LEDDEV_CNT);
222 fail_map:
223     led_unmap();
224     return -EIO;
225 }
226
227 /*
228 * @description      : platform 驱动的 remove 函数
229 * @param - dev      : platform 设备

```

```

230 * @return      : 0, 成功;其他负值,失败
231 */
232 static int led_remove(struct platform_device *dev)
233 {
234     led_unmap();                /* 取消映射      */
235     cdev_del(&leddev.cdev);     /* 删除 cdev  */
236     unregister_chrdev_region(leddev.devid, LEDDEV_CNT);
237     device_destroy(leddev.class, leddev.devid); /* 注销设备 */
238     class_destroy(leddev.class); /* 注销类      */
239     return 0;
240 }
241
242 /* platform 驱动结构体 */
243 static struct platform_driver led_driver = {
244     .driver      = {
245         .name    = "rk3568-led", /* 驱动名字,用于和设备匹配 */
246     },
247     .probe      = led_probe,
248     .remove     = led_remove,
249 };
250
251 /*
252 * @description  : 驱动模块加载函数
253 * @param        : 无
254 * @return       : 无
255 */
256 static int __init leddriver_init(void)
257 {
258     return platform_driver_register(&led_driver);
259 }
260
261 /*
262 * @description  : 驱动模块卸载函数
263 * @param        : 无
264 * @return       : 无
265 */
266 static void __exit leddriver_exit(void)
267 {
268     platform_driver_unregister(&led_driver);
269 }
270
271 module_init(leddriver_init);
272 module_exit(leddriver_exit);
    
```

```
273 MODULE_LICENSE("GPL");
274 MODULE_AUTHOR("ALIENTEK");
275 MODULE_INFO(intree, "Y");
```

leddriver.c 文件内容就是按照示例代码 18.2.2.5 的 platform 驱动模板编写的。

第 88~128 行, 传统的字符设备驱动。

第 135~225 行, probe 函数, 当设备和驱动匹配以后此函数就会执行, 当匹配成功以后会在终端上输出“led driver and device has matched!”。在 probe 函数里面初始化 LED、注册字符设备驱动。也就是将原来在驱动加载函数里面做的工作全部放到 probe 函数里面完成。

第 232~240 行, remove 函数, 当卸载 platform 驱动的时候此函数就会执行。在此函数里面释放内存、注销字符设备等。也就是将原来驱动卸载函数里面的工作全部都放到 remove 函数中完成。

第 243~249 行, platform_driver 驱动结构体, 注意 name 字段为“rk3568-led”, 和我们在 leddevice.c 文件里面设置的设备 name 字段一致。

第 256~259 行, 驱动模块加载函数, 在此函数里面通过 platform_driver_register 向 Linux 内核注册 led_driver 驱动。

第 266~269 行, 驱动模块卸载函数, 在此函数里面通过 platform_driver_unregister 从 Linux 内核卸载 led_driver 驱动。

18.4.2 测试 APP 编写

测试 APP 的内容很简单, 就是打开和关闭 LED 灯, 新建 ledApp.c 这个文件, 然后在里面输入如下内容:

示例代码 18.4.2.1 ledApp.c 文件代码段

```
1 #include "stdio.h"
2 #include "unistd.h"
3 #include "sys/types.h"
4 #include "sys/stat.h"
5 #include "fcntl.h"
6 #include "stdlib.h"
7 #include "string.h"
8 /*****
9 Copyright © ALIENTEK Co., Ltd. 1998-2029. All rights reserved.
10 文件名      : ledApp.c
11 作者        : 正点原子 Linux 团队
12 版本        : V1.0
13 描述        : platform 驱动测试 APP。
14 其他        : 无
15 使用方法    : ./ledApp /dev/platled 0 关闭 LED
16             : ./ledApp /dev/platled 1 打开 LED
17 论坛        : www.openedv.com
18 日志        : 初版 V1.0 2019/8/16 正点原子 Linux 团队创建
19 *****/
20 #define LEDOFF 0
21 #define LEDON 1
```

```

22
23 /*
24 * @description      : main 主程序
25 * @param - argc     : argv 数组元素个数
26 * @param - argv     : 具体参数
27 * @return           : 0 成功;其他 失败
28 */
29 int main(int argc, char *argv[])
30 {
31     int fd, retvalue;
32     char *filename;
33     unsigned char databuf[1];
34
35     if(argc != 3){
36         printf("Error Usage!\r\n");
37         return -1;
38     }
39
40     filename = argv[1];
41     /* 打开 led 驱动 */
42     fd = open(filename, O_RDWR);
43     if(fd < 0){
44         printf("file %s open failed!\r\n", argv[1]);
45         return -1;
46     }
47
48     databuf[0] = atoi(argv[2]); /* 要执行的操作: 打开或关闭 */
49     retvalue = write(fd, databuf, sizeof(databuf));
50     if(retvalue < 0){
51         printf("LED Control Failed!\r\n");
52         close(fd);
53         return -1;
54     }
55
56     retvalue = close(fd); /* 关闭文件 */
57     if(retvalue < 0){
58         printf("file %s close failed!\r\n", argv[1]);
59         return -1;
60     }
61     return 0;
62 }
    
```

ledApp.c 文件内容很简单, 就是控制 LED 灯的亮灭, 和第四十一章的测试 APP 基本一致, 这里就不重复讲解了。

18.5 运行测试

18.5.1 编译驱动程序和测试 APP

1、编译驱动程序

编写 Makefile 文件，本章实验的 Makefile 文件和第五章实验基本一样，只是将 obj-m 变量的值改为“leddevice.o leddriver.o”，Makefile 内容如下所示：

示例代码 18.5.1.1 Makefile 文件

```

1  KERNELDIR := /home/alientek/rk3568_linux_sdk/kernel
.....
4  obj-m := leddevice.o
5  obj-m += leddriver.o
.....
12 clean:
13 $(MAKE) -C $(KERNELDIR) M=$(CURRENT_PATH) clean
    
```

第 4, 5 行，设置 obj-m 变量的值为“leddevice.o leddriver.o”。

输入如下命令编译出驱动模块文件：

```
make ARCH=arm64 //ARCH=arm64 必须指定，否则编译会失败
```

编译成功以后就会生成一个名为“leddevice.ko leddriver.ko”的驱动模块文件。

2、编译测试 APP

输入如下命令编译测试 ledApp.c 这个测试程序：

```
/opt/atk-dlrk356x-toolchain/bin/aarch64-buildroot-linux-gnu-gcc ledApp.c -o ledApp
```

编译成功以后就会生成 ledApp 这个应用程序。

18.4.2 运行测试

在 Ubuntu 中将上一小节编译出来的 leddevice.ko、leddriver.ko 和 ledApp 这三个文件通过 adb 命令发送到开发板的/lib/modules/4.19.232 目录下，命令如下：

```
adb push leddevice.ko leddriver.ko ledApp /lib/modules/4.19.232
```

发送成功以后进入到开发板目录 lib/modules/4.19.232 中，输入如下命令加载 leddevice.ko 设备模块和 leddriver.ko 这个驱动模块：

```

depmod           //第一次加载驱动的时候需要运行此命令
modprobe leddevice //加载设备模块
modprobe leddriver //加载驱动模块
    
```

根文件系统中/sys/bus/platform/目录下保存着当前板子 platform 总线下的设备和驱动，其中 devices 子目录为 platform 设备，drivers 子目录为 platform 驱动。进入/sys/bus/platform/devices/目录，查看我们的设备是否存在，我们在 leddevice.c 中设置设备的 name 字段为“rk3568-led”，因此肯定在/sys/bus/platform/devices/目录下存在一个名字“rk3568-led”的文件，否则说明我们的设备模块加载失败，结果如图 18.4.2.1 所示：

fdef0000.i2c	pcie30-avdd1v8
fdef0800.iommu	pinctrl
fdf40000.rkvenc	psci
fdf40f00.iommu	pwm-fan
fdf80200.rkvdec	reg-dummy
fdf80800.iommu	regulatorv_0
fdff0000.rkisp	rk3568-led
fdff1a00.iommu	rk805-pinctrl
fe000000.dwmmc	rk805-pwrkey
fe010000.ethernet	rk808-clkout
fe040000.vop	rk808-regulator

设备

图 18.4.2.1 rk3568-led 设备

同理，查看/sys/bus/platform/drivers/目录，看一下驱动是否存在，我们在 leddriver.c 中设置 name 字段为“rk3568-led”，因此会在/sys/bus/platform/drivers/目录下存在名为“rk3568-led”这个文件，结果如图 18.4.2.2 所示：

dwc2	rga2	rockchip-lvds
dwc3	rk3328-codec	rockchip-mipi-dphy-rx
dwc3-of-simple	rk3399-rt5651-rk628	rockchip-nocp
dw-hdmi-cec	rk3568-led	rockchip-otp
dw-hdmi-hdcp	rk3x-i2c	rockchip-pcie-phy
dw-hdmi-i2s-audio	rk618-cru	rockchip-pdm
dwhdmi-rockchip	rk618-dsi	rockchip-pinctrl
dw-mipi-dsi	rk618-hdmi	rockchip-pm
dw_mmc	rk618-lvds	rockchip-pm-domain
dwmmc_rockchip	rk618-rgb	rockchip-pvtm
dw_wdt	rk618-scaler	rockchip-pwm
ehci-platform	rk618-vif	rockchip-rqa

驱动

图 18.4.2.2 rk3568-led 驱动

驱动模块和设备模块加载成功以后 platform 总线就会进行匹配，当驱动和设备匹配成功以后就会输出如图 18.4.2.3 所示一行语句：

```
root@ATK-DLRK356X:/lib/modules/4.19.232# modprobe leddevice
root@ATK-DLRK356X:/lib/modules/4.19.232# modprobe leddriver
[12787.775088] led driver and device has matched!
```

驱动和设备匹配成功

图 18.4.2.3 驱动和设备匹配成功

驱动和设备匹配成功以后就可以测试 LED 灯驱动了，输入如下命令打开 LED 灯：

```
./ledApp /dev/platled 1 //打开 LED 灯
```

在输入如下命令关闭 LED 灯：

```
./ledApp /dev/platled 0 //关闭 LED 灯
```

观察一下 LED 灯能否打开和关闭，如果可以的话就说明驱动工作正常，如果要卸载驱动的话输入如下命令即可：

```
rmmod leddevice.ko
rmmod leddriver.ko
```

第十九章 设备树下的 platform 驱动编写

上一章我们详细的讲解了 Linux 下的驱动分离与分层，以及总线、设备和驱动这样的驱动框架。基于总线、设备和驱动这样的驱动框架，Linux 内核提出来 platform 这个虚拟总线，相应的也有 platform 设备和 platform 驱动。上一章我们讲解了传统的、未采用设备树的 platform 设备和驱动编写方法。最新的 Linux 内核已经支持了设备树，因此在设备树下如何编写 platform 驱动就显得尤为重要，本章我们就来学习一下如何在设备树下编写 platform 驱动。

19.1 设备树下的 platform 驱动简介

platform 驱动框架分为总线、设备和驱动，其中总线不需要我们这些驱动程序员去管理，这个是 Linux 内核提供的，我们在编写驱动的时候只要关注于设备和驱动的具体实现即可。在没有设备树的 Linux 内核下，我们需要分别编写并注册 `platform_device` 和 `platform_driver`，分别代表设备和驱动。在使用设备树的时候，设备的描述被放到了设备树中，因此 `platform_device` 就不需要我们去编写了，我们只需要实现 `platform_driver` 即可。

19.1.1 创建设备的 pinctrl 节点

在 platform 驱动框架下必须使用 `pinctrl` 来配置引脚复用功能。我们以本章实验需要用到的 LED0 为例，编写 LED0 引脚的 `pinctrl` 配置。打开 `rk3568-pinctrl.dtsi` 文件，RK3568 的所有引脚 `pinctrl` 配置都是在这个文件里面完成的，在 `pinctrl` 节点下添加如下所示内容：

示例代码 19.1.1.1 GPIO 的 pinctrl 配置

```

1 led-gpios{
2     /omit-if-no-ref/
3     led_gpio: led-pin {
4         rockchip,pins =
5             <0 RK_PC0 RK_FUNC_GPIO &pcfg_pull_none>;
6     };
7 };
    
```

示例代码 19.1.1.1 中的第 3 行的 `led_gpio` 节点就是 LED 的 `pinctrl` 配置，把 `GPIO0_C0` 端口复用为 GPIO 功能。

19.1.2 在设备树中创建设备节点

接下来要在设备树中创建设备节点来描述设备信息，重点是要设置好 `compatible` 属性的值，因为 platform 总线需要通过设备节点的 `compatible` 属性值来匹配驱动！这点要切记。修改 10.4.2 小节中我们创建的 `gpioled` 节点，修改以后的内容如下：

示例代码 19.1.2.1 gpioled 设备节点

```

1 gpioled {
2     compatible = "alientek,led";
3     pinctrl-names = "alientek,led";
4     pinctrl-0 = <&led_gpio>;
5     led-gpio = <&gpio0 RK_PC0 GPIO_ACTIVE_HIGH>;
6     status = "okay";
7 };
    
```

第 2 行的 `compatible` 属性值为“alientek,led”，因此一会在编写 platform 驱动的时候 `of_match_table` 属性表中要有“alientek,led”。

第 4 行里，`pinctrl-0` 属性设置 LED 的 PIN 对应的 `pinctrl` 节点，也就是我们在示例代码 19.1.1.1 中编写的 `led_gpio`。

19.1.3 编写 platform 驱动的时候要注意兼容属性

上一章已经详细的讲解过了，在使用设备树的时候 platform 驱动会通过 of_match_table 来保存兼容性值，也就是表明此驱动兼容哪些设备。所以，of_match_table 将会尤为重要，比如本例程的 platform 驱动中 platform_driver 就可以按照如下所示设置：

示例代码 19.1.3.1 of_match_table 匹配表的设置

```
1  static const struct of_device_id led_of_match[] = {
2      { .compatible = "alientek,led" }, /* 兼容属性 */
3      { /* Sentinel */ }
4  };
5
6  MODULE_DEVICE_TABLE(of, led_of_match);
7
8  static struct platform_driver led_platform_driver = {
9      .driver = {
10         .name      = "rk3568-led",
11         .of_match_table = led_of_match,
12     },
13     .probe      = led_probe,
14     .remove     = led_remove,
15 };
```

第 1~4 行，of_device_id 表，也就是驱动的兼容表，是一个数组，每个数组元素为 of_device_id 类型。每个数组元素都是一个兼容属性，表示兼容的设备，一个驱动可以跟多个设备匹配。这里我们仅仅匹配了一个设备，那就是示例代码 19.1.2 中创建的 gpioled 这个设备。第 2 行的 compatible 值为“alientek,led”，驱动中的 compatible 属性和设备中的 compatible 属性相匹配，因此驱动中对应的 probe 函数就会执行。注意第 3 行是一个空元素，在编写 of_device_id 的时候最后一个元素一定要为空！

第 6 行，通过 MODULE_DEVICE_TABLE 声明一下 led_of_match 这个设备匹配表。

第 11 行，设置 platform_driver 中的 of_match_table 匹配表为上面创建的 leds_of_match，至此我们就设置好了 platform 驱动的匹配表了。

最后就是编写驱动程序，基于设备树的 platform 驱动和上一章无设备树的 platform 驱动基本一样，都是当驱动和设备匹配成功以后先根据设备树里的 pinctrl 属性设置 PIN 的电气特性再去执行 probe 函数。我们需要在 probe 函数里面执行字符设备驱动那一套，当注销驱动模块的时候 remove 函数就会执行，都是大同小异的。

19.2 检查引脚复用配置

19.2.1 检查引脚 pinctrl 配置

RK3568 的一个引脚可以复用为多种功能，比如 GPIO0_C0 这个 IO 就可以用作：GPIO，PWM1_M0，GPU_AVS 和 UART0_RX 这四个功能。我们在学习 STM32 单片机开发的时候，一个 IO 可以被多个外设使用，比如 GPIO0_C0 同时作为 PWM1_M0、UART0_RX，但是同一时刻只能用做一个功能，做 PWM1_M0 的时候就不能做 UART0_RX！在嵌入式 Linux 下，我们要

严格按照一个引脚对应一个功能来设计硬件,比如 GPIO0_C0 现在要用作 GPIO 来驱动 LED 灯,那么就不能将 GPIO0_C0 作为其他功能。

正点原子 RK3568 开发板上将 GPIO0_C0 连接到了 LED0 上,也就是将其用作普通的 GPIO,对应的 pinctrl 配置就是示例代码 19.1.1.1。但是 rk3568-pinctrl.dtsi 是瑞芯微根据自己官方开发板编写的,因此 GPIO0_C0 就可能被瑞芯微方用作其他功能,大家在 rk3568-pinctrl.dtsi 里面找到如下所示代码:

```

1778     pwm1 {
1779         /omit-if-no-ref/
1780         pwm1m0_pins: pwm1m0-pins {
1781             rockchip,pins =
1782                 /* pwm1_m0 */
1783                 <0 RK_PC0 1 &pcfg_pull_none>;
1784         };
1785
1786         /omit-if-no-ref/
1787         pwm1m1_pins: pwm1m1-pins {
1788             rockchip,pins =
1789                 /* pwm1_m1 */
1790                 <0 RK_PB5 4 &pcfg_pull_none>;
1791         };
1792     };

```

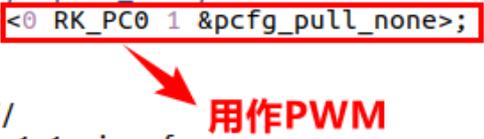


图 19.2.1.1 acodec_pins 节点

从图 19.2.1.1 可以看出,瑞芯微官方将 GPIO0_C0 复用为 ACODEC,前面说了,一个 IO 只能复用为一个功能,所以如果 pwm1 使能的话,我们就不能再将 GPIO0_C0 复用为 GPIO 了。

同样的,继续在 rk3568-pinctrl.dtsi 文件里面查找,会发现如图 19.2.1.2 所示的地方也将 GPIO0_C0 复用为了 uart0 的 rx 功能:

```

2457     uart0 {
2458         /omit-if-no-ref/
2459         uart0_xfer: uart0-xfer {
2460             rockchip,pins =
2461                 /* uart0_rx */
2462                 <0 RK_PC0 3 &pcfg_pull_up>,
2463                 /* uart0_tx */
2464                 <0 RK_PC1 3 &pcfg_pull_up>;
2465         };
2466
2467         /omit-if-no-ref/
2468         uart0_ctsn: uart0-ctsn {
2469             rockchip,pins =
2470                 /* uart0_ctsn */
2471                 <0 RK_PC7 3 &pcfg_pull_none>;
2472         };
2473
2474         /omit-if-no-ref/
2475         uart0_rtsn: uart0-rtsn {
2476             rockchip,pins =
2477                 /* uart0_rtsn */
2478                 <0 RK_PC4 3 &pcfg_pull_none>;
2479         };
2480     };

```

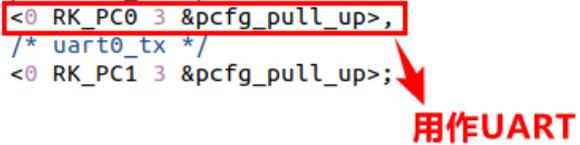


图 19.2.1.2 i2s0m0_lrck_rx 节点

图 19.2.1.2 中的 uart0-xfer 节点也将 GPIO0_C0 复用为 uart0_rx,大家继续在 rk3568-pinctrl.dtsi 文件中搜索,还会找到 GPIO0_C0 的其他复用功能。

19.2.2 检查 GPIO 占用

上一小节只是检查了一下，GPIO0_C0 这个引脚有没有被复用为多个设备，本节我们将 GPIO0_C0 复用为 GPIO。因为我们是在瑞芯微官方提供的设备树上修改的，因此还要检查一下当 GPIO0_C0 作为 GPIO 的时候，瑞芯微官方有没有将这个 GPIO 分配给其他设备。其实对于 GPIO0_C0 这个引脚来说不会的，因为瑞芯微官方没有将其复用为 GPIO，所以也就不存在说将其在作为 GPIO 分配给其他设备。但是我们在实际开发中要考虑到这一点，说不定其他的引脚就会被分配给某个设备做 GPIO，而我们没有检查，导致两个设备都用这一个 GPIO，那么肯定有一个因为申请不到 GPIO 而导致驱动无法工作。

所以当我们将一个引脚用作 GPIO 的时候，一定要检查一下当前设备树里面是否有其他设备也使用到了这个 GPIO，保证设备树中只有一个设备树在使用这个 GPIO。

19.3 硬件原理图分析

本实验的硬件原理参考 6.2 小节即可。

19.4 实验程序编写

本实验对应的例程路径为：[开发板光盘](#)→01、[程序源码](#)→Linux 驱动例程→17_dtsplatform。

本章实验我们编写基于设备树的 platform 驱动，所以需要在设备树中添加设备节点，然后我们只需要编写 platform 驱动即可。

19.4.1 修改设备树文件

首先修改设备树文件，加上我们需要的设备信息，本章我们就使用到一个 LED0。需要创建 LED0 引脚的 pinctrl 节点，这个直接使用示例代码 19.1.1.1 中的 led_gpio 节点。另外也要创建一个 LED0 设备节点，这个直接使用示例代码 19.1.2.1 中的 gpioled 设备节点。

19.4.2 platform 驱动程序编写

设备已经准备好了，接下来就要编写相应的 platform 驱动了，新建名为“17_dtsplatform”的文件夹，然后在 17_dtsplatform 文件夹里面创建 vscode 工程，工作区命名为“dtsplatform”。新建名为 leddriver.c 的驱动文件，在 leddriver.c 中输入如下所示内容：

示例代码 19.4.2.1 leddriver.c 文件代码段

```
1 #include <linux/types.h>
2 #include <linux/kernel.h>
3 #include <linux/delay.h>
4 #include <linux/ide.h>
5 #include <linux/init.h>
6 #include <linux/module.h>
7 #include <linux/errno.h>
8 #include <linux/gpio.h>
9 #include <linux/cdev.h>
10 #include <linux/device.h>
11 #include <linux/of_gpio.h>
12 #include <linux/semaphore.h>
```

```

13 #include <linux/timer.h>
14 #include <linux/irq.h>
15 #include <linux/wait.h>
16 #include <linux/poll.h>
17 #include <linux/fs.h>
18 #include <linux/fcntl.h>
19 #include <linux/platform_device.h>
20 // #include <asm/mach/map.h>
21 #include <asm/uaccess.h>
22 #include <asm/io.h>
23
24
25 #define LEDDEV_CNT      1          /* 设备号长度 */
26 #define LEDDEV_NAME    "dtsplatled" /* 设备名字 */
27 #define LEDOFF         0
28 #define LEDON          1
29
30 /* leddev 设备结构体 */
31 struct leddev_dev{
32     dev_t devid;          /* 设备号 */
33     struct cdev cdev;    /* cdev */
34     struct class *class; /* 类 */
35     struct device *device; /* 设备 */
36     struct device_node *node; /* LED 设备节点 */
37     int gpio_led;        /* LED 灯 GPIO 标号 */
38 };
39
40 struct leddev_dev leddev; /* led 设备 */
41
42 /*
43  * @description : LED 打开/关闭
44  * @param - sta : LEDON(0) 打开 LED, LEDOFF(1) 关闭 LED
45  * @return      : 无
46  */
47 void led_switch(u8 sta)
48 {
49     if (sta == LEDON )
50         gpio_set_value(leddev.gpio_led, 1);
51     else if (sta == LEDOFF)
52         gpio_set_value(leddev.gpio_led, 0);
53 }
54
55 static int led_gpio_init(struct device_node *nd)

```

```

56 {
57     int ret;
58
59     /* 从设备树中获取 GPIO */
60     leddev.gpio_led = of_get_named_gpio(nd, "led-gpio", 0);
61     if(!gpio_is_valid(leddev.gpio_led)) {
62         printk(KERN_ERR "leddev: Failed to get led-gpio\n");
63         return -EINVAL;
64     }
65
66     /* 申请使用 GPIO */
67     ret = gpio_request(leddev.gpio_led, "LED");
68     if (ret) {
69         printk(KERN_ERR "led: Failed to request led-gpio\n");
70         return ret;
71     }
72
73     /* 将 GPIO 设置为输出模式并设置 GPIO 初始电平状态 */
74     gpio_direction_output(leddev.gpio_led,0);
75
76     return 0;
77 }
78
79 /*
80  * @description   : 打开设备
81  * @param - inode : 传递给驱动的 inode
82  * @param - filp  : 设备文件, file 结构体有个叫做 private_data 的成员变量
83  *                  一般在 open 的时候将 private_data 指向设备结构体。
84  * @return        : 0 成功;其他 失败
85  */
86 static int led_open(struct inode *inode, struct file *filp)
87 {
88     return 0;
89 }
90
91 /*
92  * @description   : 向设备写数据
93  * @param - filp  : 设备文件, 表示打开的文件描述符
94  * @param - buf   : 要写给设备写入的数据
95  * @param - cnt   : 要写入的数据长度
96  * @param - offt  : 相对于文件首地址的偏移
97  * @return        : 写入的字节数, 如果为负值, 表示写入失败
98  */

```

```

99 static ssize_t led_write(struct file *filp, const char __user *buf,
size_t cnt, loff_t *offt)
100 {
101     int retvalue;
102     unsigned char databuf[1];
103     unsigned char ledstat;
104
105     retvalue = copy_from_user(databuf, buf, cnt);
106     if(retvalue < 0) {
107         printk("kernel write failed!\r\n");
108         return -EFAULT;
109     }
110
111     ledstat = databuf[0];
112     if (ledstat == LEDON) {
113         led_switch(LEDON);
114     } else if (ledstat == LEDOFF) {
115         led_switch(LEDOFF);
116     }
117     return 0;
118 }
119
120 /* 设备操作函数 */
121 static struct file_operations led_fops = {
122     .owner = THIS_MODULE,
123     .open = led_open,
124     .write = led_write,
125 };
126
127 /*
128 * @description : flatform 驱动的 probe 函数, 当驱动与
129 *               设备匹配以后此函数就会执行
130 * @param - dev : platform 设备
131 * @return      : 0, 成功;其他负值, 失败
132 */
133 static int led_probe(struct platform_device *pdev)
134 {
135     int ret;
136
137     printk("led driver and device was matched!\r\n");
138
139     /* 初始化 LED */
140     ret = led_gpio_init(pdev->dev.of_node);

```

```

141     if(ret < 0)
142         return ret;
143
144     /* 1、设置设备号 */
145     ret = alloc_chrdev_region(&leddev.devid, 0, LEDDEV_CNT,
                               LEDDEV_NAME);
146
147     if(ret < 0) {
148         pr_err("%s Couldn't alloc_chrdev_region, ret=%d\r\n",
149             LEDDEV_NAME, ret);
150         goto free_gpio;
151     }
152
153     /* 2、初始化 cdev */
154     leddev.cdev.owner = THIS_MODULE;
155     cdev_init(&leddev.cdev, &led_fops);
156
157     /* 3、添加一个 cdev */
158     ret = cdev_add(&leddev.cdev, leddev.devid, LEDDEV_CNT);
159     if(ret < 0)
160         goto del_unregister;
161
162     /* 4、创建类 */
163     leddev.class = class_create(THIS_MODULE, LEDDEV_NAME);
164     if (IS_ERR(leddev.class)) {
165         goto del_cdev;
166     }
167
168     /* 5、创建设备 */
169     leddev.device = device_create(leddev.class, NULL, leddev.devid,
170         NULL, LEDDEV_NAME);
171
172     if (IS_ERR(leddev.device)) {
173         goto destroy_class;
174     }
175
176     return 0;
177
178 destroy_class:
179     class_destroy(leddev.class);
180 del_cdev:
181     cdev_del(&leddev.cdev);
182 del_unregister:
183     unregister_chrdev_region(leddev.devid, LEDDEV_CNT);
184 free_gpio:
185     gpio_free(leddev.gpio_led);
    
```

```

181     return -EIO;
182 }
183
184 /*
185  * @description   : platform 驱动的 remove 函数
186  * @param - dev   : platform 设备
187  * @return        : 0, 成功;其他负值,失败
188  */
189 static int led_remove(struct platform_device *dev)
190 {
191     gpio_set_value(leddev.gpio_led, 0); /* 卸载驱动的时候关闭 LED */
192     gpio_free(leddev.gpio_led);         /* 注销 GPIO */
193     cdev_del(&leddev.cdev);             /* 删除 cdev */
194     unregister_chrdev_region(leddev.devid, LEDDEV_CNT);
195     device_destroy(leddev.class, leddev.devid); /* 注销设备 */
196     class_destroy(leddev.class);         /* 注销类 */
197     return 0;
198 }
199
200 /* 匹配列表 */
201 static const struct of_device_id led_of_match[] = {
202     { .compatible = "alientek,led" },
203     { /* Sentinel */ }
204 };
205
206 MODULE_DEVICE_TABLE(of, led_of_match);
207
208 /* platform 驱动结构体 */
209 static struct platform_driver led_driver = {
210     .driver = {
211         .name = "rk3568-led",           /* 驱动名字,用于和设备匹配 */
212         .of_match_table = led_of_match, /* 设备树匹配表 */
213     },
214     .probe = led_probe,
215     .remove = led_remove,
216 };
217
218 /*
219  * @description   : 驱动模块加载函数
220  * @param        : 无
221  * @return       : 无
222  */
223 static int __init leddriver_init(void)

```

```

224 {
225     return platform_driver_register(&led_driver);
226 }
227
228 /*
229 * @description   : 驱动模块卸载函数
230 * @param         : 无
231 * @return        : 无
232 */
233 static void __exit leddriver_exit(void)
234 {
235     platform_driver_unregister(&led_driver);
236 }
237
238 module_init(leddriver_init);
239 module_exit(leddriver_exit);
240 MODULE_LICENSE("GPL");
241 MODULE_AUTHOR("ALIENTEK");
242 MODULE_INFO(intree, "Y");
    
```

代码中以前讲过的知识点这里就不再重述了!

第 55~77 行, 自定义函数 `led_gpio_init`, 该函数的参数是 `struct device_node` 类型的指针, 也就是 `led` 对应的设备节点, 当调用函数的时候传递进来。

第 133~182 行, `platform` 下的 `probe` 函数: `led_probe`, 当设备树中的设备节点与驱动之间匹配成功会先去初始化 `pinctrl` 里面配置的 IO, 也就是根据示例代码 19.1.1.1 中的属性进行配置, 然后再执行 `probe` 函数, 第 140 行调用 `led_gpio_init` 函数时, 将 `pdev->dev.of_node` 作为参数传递到函数中, `platform_device` 结构体中内置了一个 `device` 结构体类型的成员变量 `dev`。在 `device` 结构体中定义了一个 `device_node` 类型的指针变量 `of_node`, 使用设备树的情况下, 当匹配成功之后, `of_node` 会指向设备树中定义的节点, 所以在这里我们不需要通过调用 `of_find_node_by_path("/gpioled")` 函数得到 `led` 的节点。我们原来在驱动加载函数里面做的工作现在全部放到 `probe` 函数里面完成。

第 189~198 行, `platform` 下的 `remove` 函数: `led_remove`, 当 `platform` 驱动模块被卸载时此函数就会执行。在此函数里面释放内存、注销字符设备等, 也就是将原来驱动卸载函数里面的工作全部都放到 `remove` 函数中完成。

第 201~204 行, 匹配表, 描述了此驱动都和什么样的设备匹配, 第 202 行添加了一条值为 `"alientek,led"` 的 `compatible` 属性值, 当设备树中某个设备节点的 `compatible` 属性值也为 `"alientek,led"` 的时候就会与此驱动匹配。

第 209~216 行, `platform_driver` 驱动结构体变量 `led_driver`, 211 行设置这个 `platform` 驱动的名字为 `"rk3568-led"`, 因此, 当驱动加载成功以后就会在 `/sys/bus/platform/drivers/` 目录下存在一个名为 `"rk3568-led"` 的文件。第 212 行绑定 `platform` 驱动的 `of_match_table` 表。

第 223~226 行, `platform` 驱动模块入口函数, 在此函数里面通过 `platform_driver_register` 向 Linux 内核注册一个 `platform` 驱动 `led_driver`。

第 233~236 行, `platform` 驱动驱动模块出口函数, 在此函数里面通过 `platform_driver_unregister` 从 Linux 内核卸载一个 `platform` 驱动 `led_driver`。

19.4.3 编写测试 APP

测试 APP 就直接使用上一章 18.4.2 小节编写的 ledApp.c 即可。

19.5 运行测试

19.5.1 编译驱动程序和测试 APP

1、编译驱动程序

编写 Makefile 文件，本章实验的 Makefile 文件和第五章实验基本一样，只是将 obj-m 变量的值改为 “leddriver.o”，Makefile 内容如下所示：

示例代码 19.5.1.1 Makefile 文件

```

1  KERNELDIR := /home/alientek/rk3568_linux_sdk/kernel
.....
4  obj-m := leddriver.o
.....
11 clean:
12 $(MAKE) -C $(KERNELDIR) M=$(CURRENT_PATH) clean
    
```

第 4 行，设置 obj-m 变量的值为 “leddriver.o”。

输入如下命令编译出驱动模块文件：

```
make ARCH=arm64 //ARCH=arm64 必须指定，否则编译会失败
```

编译成功以后就会生成一个名为 “leddriver.o” 的驱动模块文件。

2、编译测试 APP

测试 APP 直接使用上一章的 ledApp 这个测试软件即可。

19.5.2 运行测试

在 Ubuntu 中将上一小节编译出来的 leddriver.ko 通过 adb 命令发送到开发板的 /lib/modules/4.19.232 目录下，命令如下：

```
adb push leddriver.ko /lib/modules/4.19.232
```

发送成功以后进入到开发板目录 lib/modules/4.19.232 中，输入如下命令加载 leddriver.ko 这个驱动模块：

```
depmod //第一次加载驱动的时候需要运行此命令
modprobe leddriver //加载驱动模块
```

驱动模块加载完成以后到 /sys/bus/platform/drivers/ 目录下查看驱动是否存在，我们在 leddriver.c 中设置 name 字段为 “rk3568-led”，因此会在 /sys/bus/platform/drivers/ 目录下存在名为 “rk3568-led” 这个文件，结果如图 19.5.2.1 所示：

dwc2	rga2	rockchip-lvds
dwc3	rk3328-codec	rockchip-mipi-dphy-rx
dwc3-of-simple	rk3399-rt5651-rk628	rockchip-nocp
dw-hdmi-cec	rk3568-led	rockchip-otp
dw-hdmi-hdcp	rk3x-i2c	rockchip-pcie-phy
dw-hdmi-i2s-audio	rk618-cru	rockchip-pdm
dwhdmi-rockchip	rk618-dsi	rockchip-pinctrl
dw-mipi-dsi	rk618-hdmi	rockchip-pm
dw_mmc	rk618-lvds	rockchip-pm-domain
dwmmc-rockchip	rk618-rgb	rockchip-pvtm
dw_wdt	rk618-scaler	rockchip-pwm
ehci-platform	rk618-vif	rockchip-rga

图 19.5.2.1 rk3568-led 驱动

同理, 在/sys/bus/platform/devices/目录下也存在 led 的设备文件, 也就是设备树中 gpioled 这个节点, 如图 19.5.2.2 所示:

fdd60000.gpio	fe870000.csi2-dphy-hw
fdd70020.pwm	fe8a0000.usb2-phy
fdd90000.power-management	fe8b0000.usb2-phy
fdd90000.power-management:power-controller	fiq-debugger
fde00000.pvtm	fiq_debugger.0
fde40000.npu	firmware:optee
fde4b000.iommu	firmware:scmi
fde60000.gpu	firmware:sdei
fde80000.pvtm	'Fixed MDIO bus.0'
fde90000.pvtm	gpioled
fdea0400.vdpu	gpio-regulator
fdea0800.iommu	hdmi-audio-codec.2.auto
fdeb0000.rk_rga	hdmi-sound

图 19.5.2.2 gpioled 设备

驱动和模块都存在, 当驱动和设备匹配成功以后就会输出如图 19.5.2.3 所示一行语句:

```
root@ATK-DLRK356X:/lib/modules/4.19.232# modprobe leddriver
[ 475.001716] led driver and device was matched!
```

图 19.5.2.3 驱动和设备匹配成功

驱动和设备匹配成功以后就可以测试 LED 灯驱动了, 输入如下命令打开 LED 灯:

```
./ledApp /dev/dtsplatled 1 //打开 LED 灯
```

在输入如下命令关闭 LED 灯:

```
./ledApp /dev/dtsplatled 0 //关闭 LED 灯
```

观察一下 LED 灯能否打开和关闭, 如果可以的话就说明驱动工作正常, 如果要卸载驱动的话输入如下命令即可:

```
rmmod leddriver.ko
```

第二十章 Linux 自带的 LED 灯驱动实验

前面我们都是自己编写 LED 灯驱动，其实像 LED 灯这样非常基础的设备驱动，Linux 内核已经集成了。Linux 内核的 LED 灯驱动采用 platform 框架，因此我们只需要按照要求在设备树文件中添加相应的 LED 节点即可，本章我们就来学习如何使用 Linux 内核自带的 LED 驱动来驱动正点原子的 RK3568 开发板上的 LED 这个 LED 灯。

20.1 Linux 内核自带 LED 驱动使能

上一章节我们编写基于设备树的 platform LED 灯驱动, 其实 Linux 内核已经自带了 LED 灯驱动, 要使用 Linux 内核自带的 LED 灯驱动首先得先配置 Linux 内核, 使能自带的 LED 灯驱动, 输入如下命令打开 Linux 配置菜单:

```
make ARCH=arm64 menuconfig
```

按照如下路径打开 LED 驱动配置项:

```
→ Device Drivers
  → LED Support
    → LED Support for GPIO connected LEDs
```

按照上述路径, 选择“LED Support for GPIO connected LEDs”, 将其编译进 Linux 内核, 也就是在此选项上按下“Y”键, 使此选项前面变为“<*>”, 如图 20.1.1 所示:

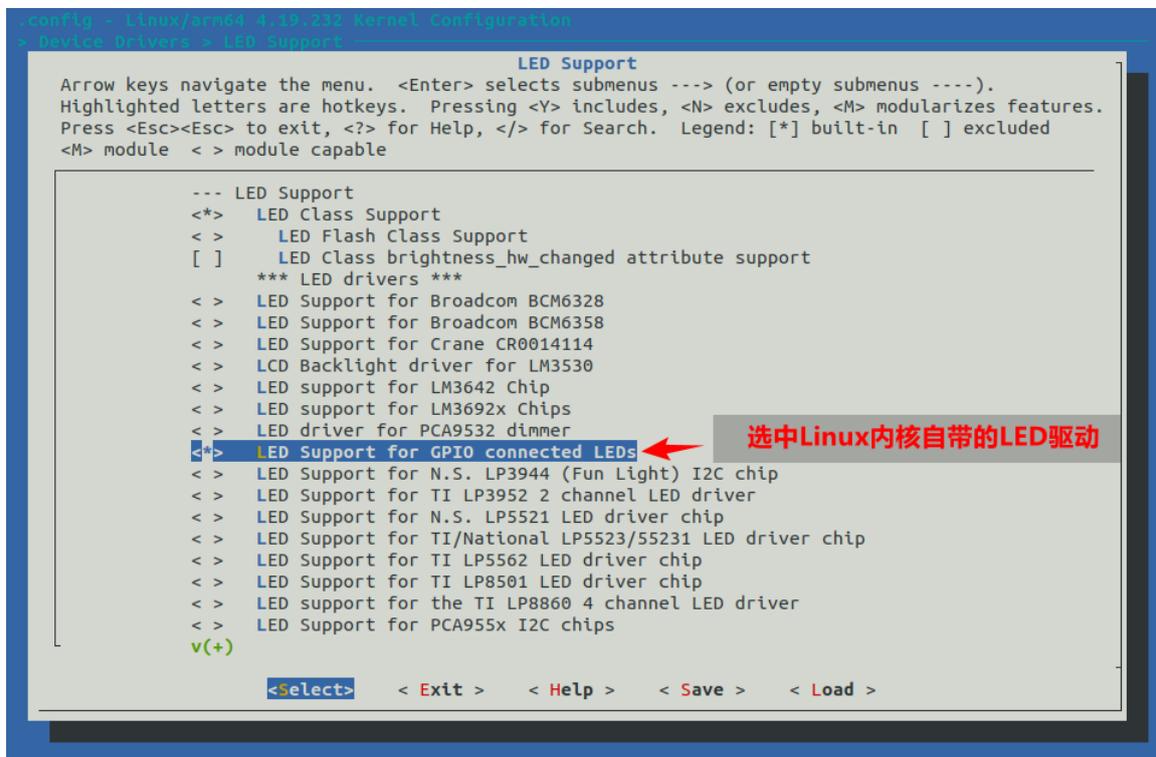


图 20.1.1 使能 LED 灯驱动

在“LED Support for GPIO connected LEDs”上按下“?”键可以打开此选项的帮助信息, 如图 20.1.2 所示:

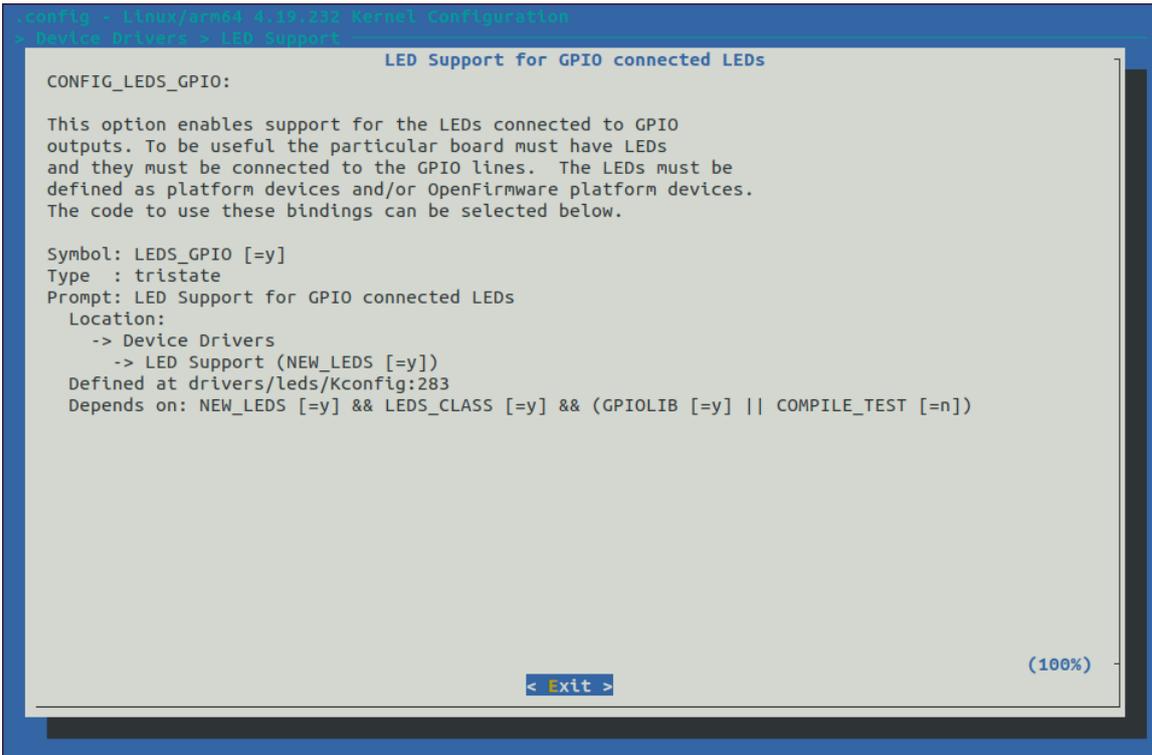


图 20.1.2 内部 LED 灯驱动帮助信息

从图 20.1.2 可以看出，把 Linux 内部自带的 LED 灯驱动编译进内核以后，CONFIG_LEDS_GPIO 就会等于 ‘y’，Linux 会根据 CONFIG_LEDS_GPIO 的值来选择如何编译 LED 灯驱动，如果为 ‘y’ 就将其编译进 Linux 内核。

配置好 Linux 内核以后退出配置界面，打开.config 文件，会找到“CONFIG_LEDS_GPIO=y”这一行，如图 20.1.3 所示：

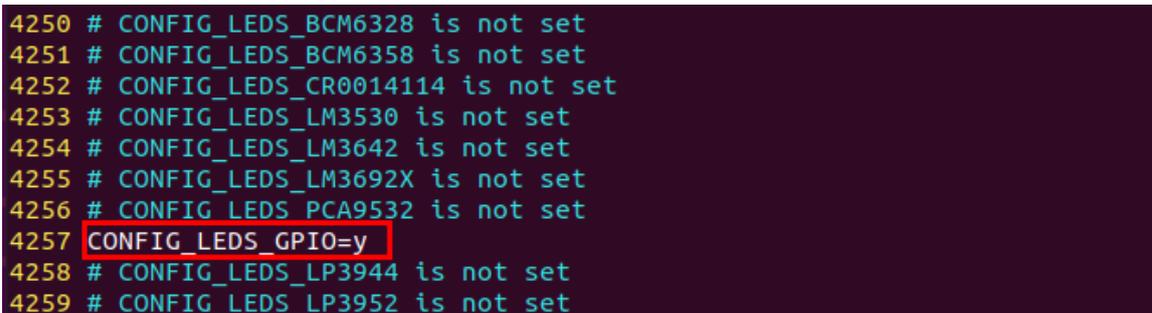


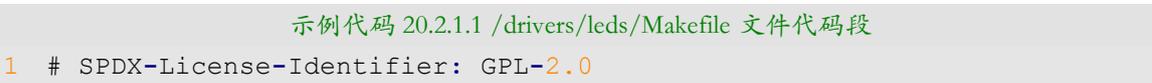
图 20.1.3 .config 文件内容

正点原子 RK3568 开发板 SDK 里面的 linux 内核默认已经使能了此驱动。

20.2 Linux 内核自带 LED 驱动简介

20.2.1 LED 灯驱动框架分析

LED 灯驱动文件为/drivers/leds/leds-gpio.c，大家可以打开/drivers/leds/Makefile 这个文件，找到如下所示内容：



```

2
3 # LED Core
4 obj-$(CONFIG_NEW_LEDS)          += led-core.o
.....
29 obj-$(CONFIG_LEDS_PCA9532)     += leds-pca9532.o
30 obj-$(CONFIG_LEDS_GPIO_REGISTER) += leds-gpio-register.o
31 obj-$(CONFIG_LEDS_GPIO)        += leds-gpio.o
32 obj-$(CONFIG_LEDS_LP3944)     += leds-lp3944.o
.....
    
```

第 31 行, 如果定义了 CONFIG_LEDS_GPIO 的话就会编译 leds-gpio.c 这个文件, 在上一小节我们选择将 LED 驱动编译进 Linux 内核, 在 .config 文件中就会有“CONFIG_LEDS_GPIO=y”这一行, 因此 leds-gpio.c 驱动文件就会被编译。

接下来我们看一下 leds-gpio.c 这个驱动文件, 找到如下所示内容:

示例代码 20.2.1.2 leds-gpio.c 文件代码段

```

215 static const struct of_device_id of_gpio_leds_match[] = {
216     { .compatible = "gpio-leds", },
217     {} ,
218 };
219
220 MODULE_DEVICE_TABLE(of, of_gpio_leds_match);
.....
267 static struct platform_driver gpio_led_driver = {
268     .probe      = gpio_led_probe,
269     .shutdown   = gpio_led_shutdown,
270     .driver     = {
271         .name    = "leds-gpio",
272         .of_match_table = of_gpio_leds_match,
273     },
274 };
275
276 module_platform_driver(gpio_led_driver);
    
```

第 215~228 行, LED 驱动的匹配表, 此表只有一个匹配项, compatible 内容为“gpio-leds”, 因此设备树中的 LED 灯设备节点的 compatible 属性值也要为“gpio-leds”, 否则设备和驱动匹配不成功, 驱动就没法工作。

第 267~274 行, platform_driver 驱动结构体变量, 可以看出, Linux 内核自带的 LED 驱动采用了 platform 框架。第 268 行可以看出 probe 函数为 gpio_led_probe, 因此当驱动和设备匹配成功以后 gpio_led_probe 函数就会执行。从 271 行可以看出, 驱动名字为“leds-gpio”, 因此会在 /sys/bus/platform/drivers 目录下存在一个名为“leds-gpio”的文件, 如图 20.2.1.1 所示:

hdmi-audio-codec	rk808-clkout	rockchip-spi
inno-hdmi-phy	rk808-regulator	rockchip-system-monitor
innohdmi-rockchip	rk808-rtc	rockchip-thermal
inno-mipi-dphy	rk817-battery	rockchip-tve
inno-video-combo-phy	rk817-charger	rockchip-typec-phy
leds-gpio	rk817-codec	rockchip-u3phy
mali	rk_codec_digital	rockchip-usb2phy
mali-utgard	rk_tiq_debugger	rockchip-usb-phy
midgard	rk_qmic	rockchip-vop
mpp-iep2	rk_gmac-dwmac	rockchip-vop2

自带的LED驱动

图 20.2.1.1 leds-gpio 驱动文件

第 276 行通过 module_platform_driver 函数向 Linux 内核注册 gpio_led_driver 这个 platform 驱动。

20.2.2 module_platform_driver 函数简析

在上一小节中我们知道 LED 驱动会采用 module_platform_driver 函数向 Linux 内核注册 platform 驱动，其实在 Linux 内核中会大量采用 module_platform_driver 来完成向 Linux 内核注册 platform 驱动的操作。module_platform_driver 定义在 include/linux/platform_device.h 文件中，为一个宏，定义如下：

示例代码 20.2.2.1 module_platform_driver 函数

```
1 #define module_platform_driver(__platform_driver) \
2     module_driver(__platform_driver, platform_driver_register, \
3     platform_driver_unregister)
```

可以看出，module_platform_driver 依赖 module_driver，module_driver 也是一个宏，定义在 include/linux/device.h 文件中，内容如下：

示例代码 20.2.2.2 module_driver 函数

```
1 #define module_driver(__driver, __register, __unregister, ...) \
2     static int __init __driver##_init(void) \
3     { \
4         return __register(&(__driver), ##__VA_ARGS__); \
5     } \
6     module_init(__driver##_init); \
7     static void __exit __driver##_exit(void) \
8     { \
9         __unregister(&(__driver), ##__VA_ARGS__); \
10    } \
11 module_exit(__driver##_exit);
```

借助示例代码 20.2.2.1 和示例代码 20.2.2.2，将：

```
module_platform_driver(gpio_led_driver)
```

展开以后就是：

```
static int __init gpio_led_driver_init(void)
{
    return platform_driver_register (&(gpio_led_driver));
}
module_init(gpio_led_driver_init);

static void __exit gpio_led_driver_exit(void)
```

```

{
    platform_driver_unregister (&(gpio_led_driver));
}
module_exit(gpio_led_driver_exit);
    
```

上面的代码不就是标准的注册和删除 platform 驱动吗？因此 module_platform_driver 函数的功能就是完成 platform 驱动的注册和删除。

20.2.3 gpio_led_probe 函数简析

当驱动和设备匹配以后 gpio_led_probe 函数就会执行，此函数主要是从设备树中获取 LED 灯的 GPIO 信息，缩减后的函数内容如下所示：

示例代码 20.2.3.1 gpio_led_probe 函数

```

1  static int gpio_led_probe(struct platform_device *pdev)
2  {
3  struct gpio_led_platform_data *pdata = dev_get_platdata (&pdev->dev);
4  struct gpio_leds_priv *priv;
5  int i, ret = 0;
6
7  if (pdata && pdata->num_leds) { /* 非设备树方式 */
..... /* 获取 platform_device 信息 */
21     }
22 } else { /* 采用设备树 */
23     priv = gpio_leds_create(pdev);
24     if (IS_ERR(priv))
25         return PTR_ERR(priv);
26 }
27
28 platform_set_drvdata(pdev, priv);
29
30 return 0;
31 }
    
```

第 23~25 行，如果使用设备树的话，使用 gpio_leds_create 函数从设备树中提取设备信息，获取到的 LED 灯 GPIO 信息保存在返回值中，gpio_leds_create 函数内容如下：

示例代码 20.2.3.2 gpio_leds_create 函数

```

1  static struct gpio_leds_priv *gpio_leds_create(struct platform_device
 *pdev)
2  {
3  struct device *dev = &pdev->dev;
4  struct fwnode_handle *child;
5  struct gpio_leds_priv *priv;
6  int count, ret;
7
8  count = device_get_child_node_count(dev);
9  if (!count)
    
```

```

10     return ERR_PTR(-ENODEV);
11
12     priv = devm_kzalloc(dev, sizeof_gpio_leds_priv(count), GFP_KERNEL);
13     if (!priv)
14         return ERR_PTR(-ENOMEM);
15
16     device_for_each_child_node(dev, child) {
17         struct gpio_led_data *led_dat = &priv->leds[priv->num_leds];
18         struct gpio_led led = {};
19         const char *state = NULL;
20         struct device_node *np = to_of_node(child);
21
22         ret = fwnode_property_read_string(child, "label", &led.name);
23         if (ret && IS_ENABLED(CONFIG_OF) && np)
24             led.name = np->name;
25         if (!led.name) {
26             fwnode_handle_put(child);
27             return ERR_PTR(-EINVAL);
28         }
29
30         led.gpiod = devm_fwnode_get_gpiod_from_child(dev, NULL, child,
31                                                     GPIOD_ASIS,
32                                                     led.name);
33         if (IS_ERR(led.gpiod)) {
34             fwnode_handle_put(child);
35             return ERR_CAST(led.gpiod);
36         }
37
38         fwnode_property_read_string(child, "linux,default-trigger",
39                                     &led.default_trigger);
40
41         if (!fwnode_property_read_string(child, "default-state",
42                                         &state)) {
43             if (!strcmp(state, "keep"))
44                 led.default_state = LEDS_GPIO_DEFSTATE_KEEP;
45             else if (!strcmp(state, "on"))
46                 led.default_state = LEDS_GPIO_DEFSTATE_ON;
47             else
48                 led.default_state = LEDS_GPIO_DEFSTATE_OFF;
49         }
50
51         if (fwnode_property_present(child, "retain-state-suspended"))
52             led.retain_state_suspended = 1;
    
```

```

53     if (fwnode_property_present(child, "retain-state-shutdown"))
54         led.retain_state_shutdown = 1;
55     if (fwnode_property_present(child, "panic-indicator"))
56         led.panic_indicator = 1;
57
58     ret = create_gpio_led(&led, led_dat, dev, np, NULL);
59     if (ret < 0) {
60         fwnode_handle_put(child);
61         return ERR_PTR(ret);
62     }
63     led_dat->cdev.dev->of_node = np;
64     priv->num_leds++;
65 }
66
67 return priv;
68 }
    
```

第 8 行, 调用 `device_get_child_node_count` 函数统计子节点数量, 一般在设备树中创建一个节点表示 LED 灯, 然后在这个节点下面为每个 LED 灯创建一个子节点。因此子节点数量也是 LED 灯的数量。

第 16 行, 遍历每个子节点, 获取每个子节点的信息。

第 30 行, 获取 LED 灯所使用的 GPIO 信息。

第 38~39 行, 获取“linux,default-trigger”属性值, 可以通过此属性设置某个 LED 灯在 Linux 系统中的默认功能, 比如作为系统心跳指示灯等等。

第 41~42 行, 获取“default-state”属性值, 也就是 LED 灯的默认状态属性。

第 58 行, 调用 `create_gpio_led` 函数创建 LED 相关的 io, 其实就是设置 LED 所使用的 io 为输出之类的。`create_gpio_led` 函数主要是初始化 `led_dat` 这个 `gpio_led_data` 结构体类型变量, `led_dat` 保存了 LED 的操作函数等内容。

关于 `gpio_led_probe` 函数就分析到这里, `gpio_led_probe` 函数主要功能就是获取 LED 灯的设备信息, 然后根据这些信息来初始化对应的 IO, 设置为输出等。

20.3 设备树节点编写

打开文档 `Documentation/devicetree/bindings/leds/leds-gpio.txt`, 此文档详细的讲解了 Linux 自带驱动对应的设备树节点该如何编写, 我们在编写设备节点的时候要注意以下几点:

①、创建一个节点表示 LED 灯设备, 比如 `dtsleds`, 如果板子上有多个 LED 灯的话每个 LED 灯都作为 `dtsleds` 的子节点。

②、`dtsleds` 节点的 `compatible` 属性值一定要为“`gpio-leds`”。

③、设置 `label` 属性, 此属性为可选, 每个子节点都有一个 `label` 属性, `label` 属性一般表示 LED 灯的名字, 比如以颜色区分的话就是 `red`、`green` 等等。

④、每个子节点必须要设置 `gpios` 属性值, 表示此 LED 所使用的 GPIO 引脚!

⑤、可以设置“`linux,default-trigger`”属性值, 也就是设置 LED 灯的默认功能, 查阅 `Documentation/devicetree/bindings/leds/common.txt` 这个文档来查看可选功能, 比如:

backlight: LED 灯作为背光。

default-on: LED 灯打开。

heartbeat: LED 灯作为心跳指示灯，可以作为系统运行提示灯。

disk-activity: LED 灯作为磁盘活动指示灯。

ide-disk: LED 灯作为硬盘活动指示灯。

timer: LED 灯周期性闪烁，由定时器驱动，闪烁频率可以修改。

⑥、可以设置“default-state”属性值，可以设置为 on、off 或 keep，为 on 的时候 LED 灯默认打开，为 off 的话 LED 灯默认关闭，为 keep 的话 LED 灯保持当前模式。

另外还有一些其他的可选属性，比如 led-sources、color、function 等属性，这些属性的用法在 Documentation/devicetree/bindings/leds/common.txt 里面有详细的讲解，大家自行查阅。

默认我们把 RK3568 开发板上的 1 个 LED 灯作为了系统运行指示灯，LED 连接到 GPIO0_C0 引脚上。LED 用作系统指示灯，名字为“work”。打开 rk3568-evb.dtsi 文件，找到如下内容：

示例代码 20.3.1 leds 设备节点

```

1  leds: leds {
2      compatible = "gpio-leds";
3      work_led: work {
4          gpios = <&gpio0 RK_PC0 GPIO_ACTIVE_HIGH>;
5          linux,default-trigger = "heartbeat";
6          status = "okay";
7      };
8  };
    
```

示例代码 20.3.1 就是正点原子出厂系统中已经编写好的 LED 灯节点，节点名字为“leds”。

第 3~7 行是开发板上的 LED，这里将 LED 用作了“heartbeat”，也就是心跳灯，因此大家烧写出厂系统会发现绿色的 LED 灯会一直闪烁。

注意，第 8 行的 status 要为“okay”，我们前面做 LED 灯实验的时候让大家把这个改为了“disabled”，这里一定要确保为“okay”！

20.4 运行测试

启动开发板，启动以后查看/sys/bus/platform/devices/leds 这个目录是否存在，如图 20.4.1 所示：

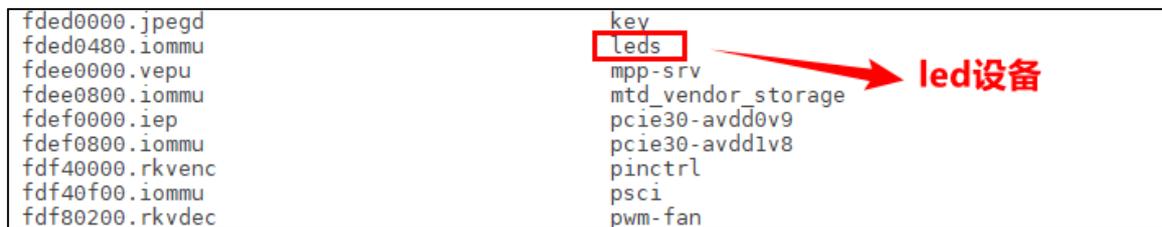


图 20.4.1 leds 目录

进入到 leds/leds 目录中，此目录中的内容如图 20.4.2 所示：

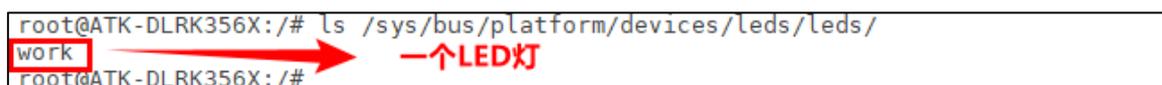


图 20.4.2 leds 目录内容

从图 20.4.2 可以看出，在 leds 目录下有一个子目录，work 就是 LED。这个子目录的名字就是我们在示例代码 20.3.1 中第 3 行设置的 label 属性值。

我们的设置究竟有没有用，最终是要通过测试才能知道的，我们以 work 灯为例，讲解一下怎么测试。首先查看一下系统中有没有“/sys/class/leds/user-led/brightness”这个文件，通过操作这两个文件即可实现 LED 的打开和关闭。

注意！由于将 LED，也就是绿色 LED 灯作为了心跳灯，因此大家使用上述命令打开和关闭会看不出来效果，必须要先禁止掉 LED 的心跳灯功能，输入如下命令：

```
echo none > /sys/class/leds/work/trigger //关闭 LED 的心跳灯功能
```

关闭心跳灯功能后就可以使用前面命令来打开和关闭 LED 了。

如果有的话就输入如下命令打开 user-led(LED1)：

```
echo 1 > /sys/class/leds/work/brightness //打开绿色 LED
```

关闭 LED1 的命令如下：

```
echo 0 > /sys/class/leds/work/brightness //关闭绿色 LED
```

如果能正常的打开和关闭 LED1 灯话就说明我们 Linux 内核自带的 LED 灯驱动工作正常。

第二十一章 Linux MISC 驱动实验

misc 的意思是混合、杂项的，因此 MISC 驱动也叫做杂项驱动，也就是当我们板子上的某些外设无法进行分类的时候就可以使用 MISC 驱动。MISC 驱动其实就是最简单的字符设备驱动，通常嵌套在 platform 总线驱动中，实现复杂的驱动，本章我们就来学习一下 MISC 驱动的编写。

21.1 MISC 设备驱动简介

所有的 MISC 设备驱动的主设备号都为 10，不同的设备使用不同的从设备号。随着 Linux 字符设备驱动的不断增长，设备号变得越来越紧张，尤其是主设备号，MISC 设备驱动就用于解决此问题。MISC 设备会自动创建 cdev，不需要像我们以前那样手动创建，因此采用 MISC 设备驱动可以简化字符设备驱动的编写。我们需要向 Linux 注册一个 miscdevice 设备，miscdevice 是一个结构体，定义在文件 include/linux/miscdevice.h 中，内容如下：

示例代码 21.1.1 miscdevice 结构体代码

```

1 struct miscdevice {
2     int minor;                /* 子设备号 */
3     const char *name;        /* 设备名字 */
4     const struct file_operations *fops; /* 设备操作集 */
5     struct list_head list;
6     struct device *parent;
7     struct device *this_device;
8     const struct attribute_group **groups;
9     const char *nodename;
10    umode_t mode;
11 };
    
```

定义一个 MISC 设备(miscdevice 类型)以后我们需要设置 minor、name 和 fops 这三个成员变量。minor 表示子设备号，MISC 设备的主设备号为 10，这个是固定的，需要用户指定子设备号，Linux 系统已经预定义了一些 MISC 设备的子设备号，这些预定义的子设备号定义在 include/linux/miscdevice.h 文件中，如下所示：

示例代码 21.1.2 预定义的 MISC 设备子设备号

```

1 #define PSMOUSE_MINOR      1
2 #define MS_BUSMOUSE_MINOR  2 /* unused */
3 #define ATIXL_BUSMOUSE_MINOR 3 /* unused */
4 /*#define AMIGAMOUSE_MINOR 4 FIXME OBSOLETE */
5 #define ATARIMOUSE_MINOR  5 /* unused */
6 #define SUN_MOUSE_MINOR    6 /* unused */
7 .....
47 #define MISC_DYNAMIC_MINOR 255
    
```

我们在使用的时候可以从这些预定义的子设备号中挑选一个，当然也可以自己定义，只要这个子设备号没有被其他设备使用接口。

name 就是此 MISC 设备名字，当此设备注册成功以后就会在/dev 目录下生成一个名为 name 的设备文件。fops 就是字符设备的操作集合，MISC 设备驱动最终是需要使用用户提供的 fops 操作集合。

当设置好 miscdevice 以后就需要使用 misc_register 函数向系统中注册一个 MISC 设备，此函数原型如下：

```
int misc_register(struct miscdevice * misc)
```

函数参数和返回值含义如下：

misc: 要注册的 MISC 设备。

返回值: 负数，失败；0，成功。

以前我们需要自己调用一堆的函数去创建设备，比如在以前的字符设备驱动中我们会使用如下几个函数完成设备创建过程：

示例代码 21.1.3 传统的创建设备过程

```

1 alloc_chrdev_region();      /* 申请设备号 */
2 cdev_init();               /* 初始化 cdev */
3 cdev_add();                /* 添加 cdev */
4 class_create();           /* 创建类 */
5 device_create();          /* 创建设备 */
    
```

现在我们可以直接使用 `misc_register` 一个函数来完成示例代码 21.1.3 中的这些步骤。当我们卸载设备驱动模块的时候需要调用 `misc_deregister` 函数来注销掉 MISC 设备，函数原型如下：

```
int misc_deregister(struct miscdevice *misc)
```

函数参数和返回值含义如下：

misc: 要注销的 MISC 设备。

返回值: 负数，失败；0，成功。

以前注销设备驱动的时候，我们需要调用一堆的函数去删除此前创建的 `cdev`、设备等等内容，如下所示：

示例代码 21.1.4 传统的删除设备的过程

```

1 cdev_del();                /* 删除 cdev */
2 unregister_chrdev_region(); /* 注销设备号 */
3 device_destroy();          /* 删除设备 */
4 class_destroy();           /* 删除类 */
    
```

现在我们只需要一个 `misc_deregister` 函数即可完成示例代码 21.1.4 中的这些工作。关于 MISC 设备驱动就讲解到这里，接下来我们就使用 `platform` 加 MISC 驱动框架来编写 LED 灯蜂鸣器驱动。

21.2 硬件原理图分析

本章实验我们只使用到正点原子的 ATK-DLRK3568 开发板上的 LED，也就是绿色的 LED 灯。

21.3 实验程序编写

本实验对应的例程路径为：[开发板光盘](#)→01、[程序源码](#)→Linux 驱动例程→18_misclcd。

本章实验我们采用 `platform` 加 `misc` 的方式编写 led 驱动，这也是实际的 Linux 驱动中很常用的方法。采用 `platform` 来实现总线、设备和驱动，`misc` 主要负责完成字符设备的创建。

21.3.1 修改设备树

本章实验我们需要用到 LED，因此需要在 `rk3568-evb.dtsi` 文件中 LED 设备节点。

1、关闭 LED1 默认功能

在上一章实验中，我们将 ATK-DLRK3568 开发板上的 LED 用 Linux 内核自带的 LED 驱动，并将其设置为“work”。此我们首先需要关闭上一章实验设置的这个 LED 功能，只需要将“work”这个节点中的 `status` 属性改为 `disabled` 即可，如图 21.3.1.1 所示：

```

162     leds: leds {
163         compatible = "gpio-leds";
164         work_led: work {
165             gpios = <&gpio0 RK_PC0 GPIO_ACTIVE_HIGH>;
166             linux,default-trigger = "heartbeat";
167             status = "disabled";
168         };
169     };
170

```

status改为disabled

图 21.3.1.1 关闭 LED1 的默认功能

2、添加 LED1 的 misc 设备节点

在 rk3568-atk-evb1-ddr4-v10.dtsi 文件的 “/” 节点下中创建一个名为 “miscled” 的子节点，miscled 子节点内容如下：

示例代码 21.3.1.1 LED1 杂项设备节点

```

1 miscled {
2     compatible = "alientek,miscled";
3     miscled-gpio = <&gpio0 RK_PC0 GPIO_ACTIVE_HIGH>;
4     status = "okay";
5 };

```

设备树修改完成，重新编译内核并烧写。

21.3.2 misc 驱动程序编写

新建名为 “18_miscled” 的文件夹，然后在 18_miscled 文件夹里面创建 vscode 工程，工作区命名为 “miscled”。新建名为 miscled.c 的驱动文件，在 miscled.c 中输入如下所示内容：

示例代码 21.3.2.1 miscled.c 文件代码段

```

1 #include <linux/types.h>
2 #include <linux/kernel.h>
3 #include <linux/delay.h>
4 #include <linux/ide.h>
5 #include <linux/init.h>
6 #include <linux/module.h>
7 #include <linux/errno.h>
8 #include <linux/gpio.h>
9 #include <linux/cdev.h>
10 #include <linux/device.h>
11 #include <linux/of.h>
12 #include <linux/of_address.h>
13 #include <linux/of_gpio.h>
14 #include <linux/platform_device.h>
15 #include <linux/miscdevice.h>
16 // #include <asm/mach/map.h>
17 #include <asm/uaccess.h>
18 #include <asm/io.h>
19
20 #define MISCLEDE_NAME "miscled" /* 名字 */

```

```

21 #define MISCLELED_MINOR      144          /* 子设备号      */
22 #define LEDOFF               0           /* 关 LED        */
23 #define LEDON                1           /* 开 LED        */
24
25 /* miscled 设备结构体 */
26 struct miscled_dev{
27     dev_t devid;                /* 设备号        */
28     struct cdev cdev;          /* cdev          */
29     struct class *class;       /* 类            */
30     struct device *device;     /* 设备          */
31     int led_gpio;             /* led 所使用的 GPIO 编号      */
32 };
33
34 struct miscled_dev miscled;    /* led 设备 */
35
36 /*
37  * @description   : beep 相关初始化操作
38  * @param - pdev : platform_device 指针, 也就是 platform 设备指针
39  * @return       : 成功返回 0, 失败返回负数
40  */
41 static int led_gpio_init(struct device_node *nd)
42 {
43     int ret;
44
45     /* 从设备树中获取 GPIO */
46     miscled.led_gpio = of_get_named_gpio(nd, "miscled-gpio", 0);
47     if(!gpio_is_valid(miscled.led_gpio)) {
48         printk("miscled: Failed to get led-gpio\n");
49         return -EINVAL;
50     }
51
52     /* 申请使用 GPIO */
53     ret = gpio_request(miscled.led_gpio, "led");
54     if(ret) {
55         printk("led: Failed to request miscled-gpio\n");
56         return ret;
57     }
58
59     /* 将 GPIO 设置为输出模式并设置 GPIO 初始化电平状态 */
60     gpio_direction_output(miscled.led_gpio, 0);
61
62     return 0;
63 }
    
```

```

64
65 /*
66 * @description   : 打开设备
67 * @param - inode : 传递给驱动的 inode
68 * @param - filp  : 设备文件, file 结构体有个叫做 private_data 的成员变量
69 *                  一般在 open 的时候将 private_data 指向设备结构体。
70 * @return        : 0 成功;其他 失败
71 */
72 static int miscled_open(struct inode *inode, struct file *filp)
73 {
74     return 0;
75 }
76
77 /*
78 * @description   : 向设备写数据
79 * @param - filp  : 设备文件, 表示打开的文件描述符
80 * @param - buf   : 要写给设备写入的数据
81 * @param - cnt   : 要写入的数据长度
82 * @param - offt  : 相对于文件首地址的偏移
83 * @return        : 写入的字节数, 如果为负值, 表示写入失败
84 */
85 static ssize_t miscled_write(struct file *filp, const char __user
*buf, size_t cnt, loff_t *offt)
86 {
87     int retvalue;
88     unsigned char databuf[1];
89     unsigned char ledstat;
90
91     retvalue = copy_from_user(databuf, buf, cnt);
92     if(retvalue < 0) {
93         printk("kernel write failed!\r\n");
94         return -EFAULT;
95     }
96
97     ledstat = databuf[0];        /* 获取状态值 */
98     if(ledstat == LEDON) {
99         gpio_set_value(miscled.led_gpio, 1);    /* 打开 LED */
100    } else if(ledstat == LEDOFF) {
101        gpio_set_value(miscled.led_gpio, 0);    /* 关闭 LED */
102    }
103    return 0;
104 }
105

```

```

106 /* 设备操作函数 */
107 static struct file_operations miscled_fops = {
108     .owner = THIS_MODULE,
109     .open = miscled_open,
110     .write = miscled_write,
111 };
112
113 /* MISC 设备结构体 */
114 static struct miscdevice led_miscdev = {
115     .minor = MISCLLED_MINOR,
116     .name = MISCLLED_NAME,
117     .fops = &miscled_fops,
118 };
119
120 /*
121  * @description : flatform 驱动的 probe 函数, 当驱动与
122  *               设备匹配以后此函数就会执行
123  * @param - dev : platform 设备
124  * @return      : 0, 成功;其他负值, 失败
125  */
126 static int miscled_probe(struct platform_device *pdev)
127 {
128     int ret = 0;
129
130     printk("led driver and device was matched!\r\n");
131
132     /* 初始化 BEEP */
133     ret = led_gpio_init(pdev->dev.of_node);
134     if(ret < 0)
135         return ret;
136
137     /* 一般情况下会注册对应的字符设备, 但是这里我们使用 MISC 设备
138      * 所以我们不需要自己注册字符设备驱动, 只需要注册 misc 设备驱动即可
139      */
140     ret = misc_register(&led_miscdev);
141     if(ret < 0){
142         printk("misc device register failed!\r\n");
143         goto free_gpio;
144     }
145
146     return 0;
147
148 free_gpio:
    
```

```

149     gpio_free(miscled.led_gpio);
150     return -EINVAL;
151 }
152
153 /*
154  * @description   : platform 驱动的 remove 函数
155  * @param - dev   : platform 设备
156  * @return        : 0, 成功;其他负值, 失败
157  */
158 static int miscled_remove(struct platform_device *dev)
159 {
160     /* 注销设备的时候关闭 LED 灯 */
161     gpio_set_value(miscled.led_gpio, 1);
162
163     /* 释放 LED */
164     gpio_free(miscled.led_gpio);
165
166     /* 注销 misc 设备 */
167     misc_deregister(&led_miscdev);
168     return 0;
169 }
170
171 /* 匹配列表 */
172 static const struct of_device_id led_of_match[] = {
173     { .compatible = "alientek,miscled" },
174     { /* Sentinel */ }
175 };
176
177 /* platform 驱动结构体 */
178 static struct platform_driver led_driver = {
179     .driver = {
180         .name = "alientek,miscled", /* 驱动名字, 用于和设备匹配 */
181         .of_match_table = led_of_match, /* 设备树匹配表 */
182     },
183     .probe = miscled_probe,
184     .remove = miscled_remove,
185 };
186
187 /*
188  * @description   : 驱动出口函数
189  * @param         : 无
190  * @return        : 无
191  */

```

```

192 static int __init miscled_init(void)
193 {
194     return platform_driver_register(&led_driver);
195 }
196
197 /*
198  * @description   : 驱动出口函数
199  * @param         : 无
200  * @return        : 无
201  */
202 static void __exit miscled_exit(void)
203 {
204     platform_driver_unregister(&led_driver);
205 }
206
207 module_init(miscled_init);
208 module_exit(miscled_exit);
209 MODULE_LICENSE("GPL");
210 MODULE_AUTHOR("ALIENTEK");
211 MODULE_INFO(intree, "Y");
    
```

第 72~111 行，标准的字符设备驱动。

第 114~118 行，MISC 设备 led_miscdev，第 115 行设置子设备号为 144，第 116 行设置设备名字为“miscled”，这样当系统启动以后就会在/dev/目录下存在一个名为“miscled”的设备文件。第 117 行，设置 MISC 设备的操作函数集合，为 file_operations 类型。

第 126~151 行，platform 框架的 probe 函数，当驱动与设备匹配以后此函数就会执行，首先在此函数中初始化 LED1 所使用的 IO。最后在 140 行通过 misc_register 函数向 Linux 内核注册 MISC 设备，也就是前面定义的 led_miscdev。

第 158~169 行，platform 框架的 remove 函数，在此函数中调用 misc_deregister 函数来注销 MISC 设备。

第 192~205，标准的 platform 驱动。

21.3.3 编写测试 APP

新建 miscledApp.c 文件，然后在里面输入如下所示内容：

示例代码 21.3.2.2 miscledApp.c 文件代码段

```

1 #include <stdio.h>
2 #include <unistd.h>
3 #include <sys/types.h>
4 #include <sys/stat.h>
5 #include <fcntl.h>
6 #include <stdlib.h>
7 #include <string.h>
8
9 /*
    
```

```

10 * @description      : main 主程序
11 * @param - argc     : argv 数组元素个数
12 * @param - argv     : 具体参数
13 * @return           : 0 成功;其他 失败
14 */
15 int main(int argc, char *argv[])
16 {
17     int fd, retvalue;
18     char *filename;
19     unsigned char databuf[1];
20
21     if(argc != 3){
22         printf("Error Usage!\r\n");
23         return -1;
24     }
25
26     filename = argv[1];
27     fd = open(filename, O_RDWR); /* 打开 led 驱动 */
28     if(fd < 0){
29         printf("file %s open failed!\r\n", argv[1]);
30         return -1;
31     }
32
33     databuf[0] = atoi(argv[2]); /* 要执行的操作: 打开或关闭 */
34     retvalue = write(fd, databuf, sizeof(databuf));
35     if(retvalue < 0){
36         printf("BEEP Control Failed!\r\n");
37         close(fd);
38         return -1;
39     }
40
41     retvalue = close(fd); /* 关闭文件 */
42     if(retvalue < 0){
43         printf("file %s close failed!\r\n", argv[1]);
44         return -1;
45     }
46     return 0;
47 }

```

miscledApp.c 文件内容和其他例程的测试 APP 基本一致, 很简单, 这里就不讲解了。

21.4 运行测试

21.4.1 编译驱动程序和测试 APP

1、编译驱动程序

编写 Makefile 文件，本章实验的 Makefile 文件和第五章实验基本一样，只是将 obj-m 变量的值改为“miscled.o”，Makefile 内容如下所示：

示例代码 21.4.1.1 Makefile 文件

```
1 KERNELDIR := /home/alientek/rk3568_linux_sdk/kernel
.....
4 obj-m := miscled.o
.....
11 clean:
12 $(MAKE) -C $(KERNELDIR) M=$(CURRENT_PATH) clean
```

第 4 行，设置 obj-m 变量的值为“miscled.o”。

输入如下命令编译出驱动模块文件：

```
make ARCH=arm64 //ARCH=arm64 必须指定，否则编译会失败
```

编译成功以后就会生成一个名为“miscled.ko”的驱动模块文件。

2、编译测试 APP

输入如下命令编译测试 miscledApp.c 这个测试程序：

```
/opt/atk-dlrk356x-toolchain/bin/aarch64-buildroot-linux-gnu-gcc miscledApp.c -o miscledApp
```

编译成功以后就会生成 miscledApp 这个应用程序。

21.4.2 运行测试

在 Ubuntu 中将上一小节编译出来的 miscled.ko 和 miscledApp 通过 adb 命令发送到开发板的/lib/modules/4.19.232 目录下，命令如下：

```
adb push miscled.ko miscledApp /lib/modules/4.19.232
```

发送成功以后进入到开发板目录 lib/modules/4.19.232 中，输入如下命令加载 miscled.ko 这个驱动模块：

```
depmod //第一次加载驱动的时候需要运行此命令
modprobe miscled //加载驱动模块
```

当驱动模块加载成功以后我们可以在/sys/class/misc 这个目录下看到一个名为“miscled”的子目录，如图 21.4.2.1 所示：

```
root@ATK-DLRK356X:/lib/modules/4.19.232# ls /sys/class/misc
cpu_dma_latency  hw_random      miscled         rfkill         vendor_storage
crypto           lightsensor    mma8452_daemon rga            vinci
fuse             loop-control   network_latency ubi_ctrl       watchdog
gyrosensor      mali0          network_throughput uhid
hdmir_hdcp1x    memory_bandwidth psensor         uinput
```

图 21.4.2.1 miscled 子目录

所有的 misc 设备都属于同一个类，/sys/class/misc 目录下就是 misc 这个类的所有设备，每个设备对应一个子目录。

驱动与设备匹配成功以后就会生成/dev/miscled 这个设备驱动文件，输入如下命令查看这个

文件的主次设备号:

```
ls /dev/miscled -l
```

结果如图 21.4.2.2 所示:

```
root@ATK-DLRK356X:/lib/modules/4.19.232# ls /dev/miscled -l
crw----- 1 root root 10, 144 Aug  4 17:41 /dev/miscled
root@ATK-DLRK356X:/lib/modules/4.19.232#
```

图 21.4.2.2 /dev/miscled 设备文件

从图 21.4.2.2 可以看出, /dev/miscled 这个设备的主设备号为 10, 次设备号为 144, 和我们驱动程序里面设置的一致。

输入如下命令打开 LED:

```
./miscledApp /dev/miscled 1 //打开 LED
```

在输入如下命令关闭 LED 灯:

```
./miscledApp /dev/miscled 0 //关闭 LED
```

观察一下 LED 能否打开和关闭, 如果可以的话就说明驱动工作正常, 如果要卸载驱动的话输入如下命令即可:

```
rmmod miscled.ko
```

第二十二章 Linux INPUT 子系统实验

按键、鼠标、键盘、触摸屏等都属于输入(input)设备, Linux 内核为此专门做了一个叫做 input 子系统的框架来处理输入事件。输入设备本质上还是字符设备, 只是在此基础上套上了 input 框架, 用户只需要负责上报输入事件, 比如按键值、坐标等信息, input 核心层负责处理这些事件。本章我们就来学习一下 Linux 内核中的 input 子系统。

22.1 input 子系统

22.1.1 input 子系统简介

input 就是输入的意思，因此 input 子系统就是管理输入的子系统，和 pinctrl、gpio 子系统一样，都是 Linux 内核针对某一类设备而创建的框架。比如按键输入、键盘、鼠标、触摸屏等等这些都属于输入设备，不同的输入设备所代表的含义不同，按键和键盘就是代表按键信息，鼠标和触摸屏代表坐标信息，因此在应用层的处理就不同，对于驱动编写者而言不需要去关心应用层的事情，我们只需要按照要求上报这些输入事件即可。为此 input 子系统分为 input 驱动层、input 核心层、input 事件处理层，最终给用户空间提供可访问的设备节点，input 子系统框架如图 22.1.1.1 所示：

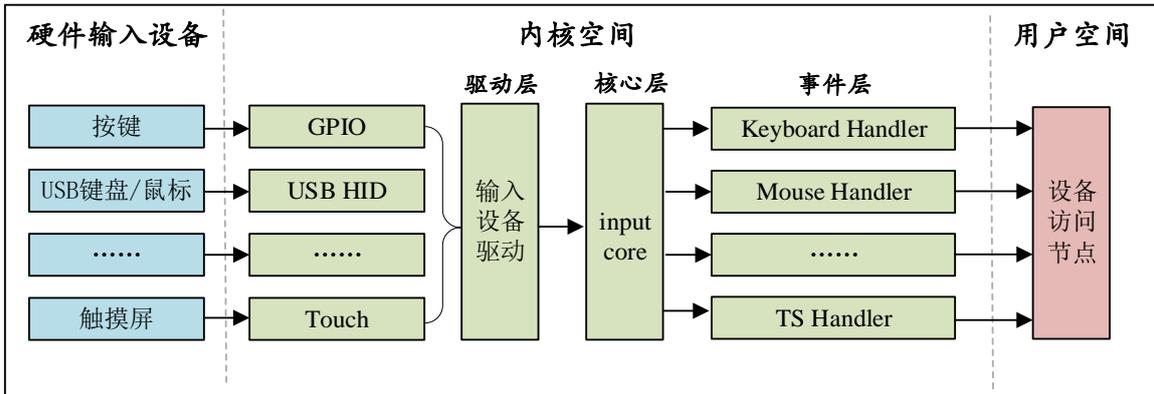


图 22.1.1.1 input 子系统结构图

图 22.1.1.1 中左边就是最底层的具体设备，比如按键、USB 键盘/鼠标等，中间部分属于 Linux 内核空间，分为驱动层、核心层和事件层，最右边的就是用户空间，所有的输入设备以文件的形式供用户应用程序使用。可以看出 input 子系统用到了我们前面讲解的驱动分层模型，我们编写驱动程序的时候只需要关注中间的驱动层、核心层和事件层，这三个层的分工如下：

驱动层：输入设备的具体驱动程序，比如按键驱动程序，向内核层报告输入内容。

核心层：承上启下，为驱动层提供输入设备注册和操作接口。通知事件层对输入事件进行处理。

事件层：主要和用户空间进行交互。

22.1.2 input 驱动编写流程

input 核心层会向 Linux 内核注册一个字符设备，大家找到 drivers/input/input.c 这个文件，input.c 就是 input 输入子系统的核心层，此文件里面有如下所示代码：

示例代码 22.1.2.1 input 核心层创建字符设备过程

```

1 struct class input_class = {
2     .name           = "input",
3     .devnode       = input_devnode,
4 };
5 .....
6 static int __init input_init(void)
7 {
8     int err;

```

```

9
10  err = class_register(&input_class);
11  if (err) {
12      pr_err("unable to register input_dev class\n");
13      return err;
14  }
15
16  err = input_proc_init();
17  if (err)
18      goto fail1;
19
20  err = register_chrdev_region(MKDEV(INPUT_MAJOR, 0),
21                              INPUT_MAX_CHAR_DEVICES, "input");
22  if (err) {
23      pr_err("unable to register char major %d", INPUT_MAJOR);
24      goto fail2;
25  }
26
27  return 0;
28
29 fail2: input_proc_exit();
30 fail1: class_unregister(&input_class);
31 return err;
32 }

```

第 10 行，注册一个 input 类，这样系统启动以后就会在 /sys/class 目录下有一个 input 子目录，如图 22.1.2.1 所示：

```

root@ATK-DLRK356X:/lib/modules/4.19.232# ls /sys/class
android_usb  extcon      mmc_host    rfkill      tpm
ata_device   gpio        mpp_class   rkwifi      tpmrm
ata_link     graphics    mtd         rtc         tty
ata_port     hidraw      net         scsi_device ubi
backlight    hwmon       nvme        scsi_disk   udc
bdi          i2c-adapter nvme-subsystem scsi_host   usbmon
block        i2c-dev     pci_bus     sensor_class vc
bluetooth    ieee80211   phy         sound       video4linux
bsg          input       power_supply spidev      vtconsole
devcoredump iommu       ppp         spi_host    wakeup
devfreq      leds        pps         spi_master  watchdog
devfreq-event mdio_bus    ptp         spi_transport wwan_5g
dma          mem         pwm         tee         zram-control
drm          misc        regulator   thermal
root@ATK-DLRK356X:/lib/modules/4.19.232#

```

图 22.1.2.1 input 类

第 20~21 行，注册一个字符设备，主设备号为 INPUT_MAJOR，INPUT_MAJOR 定义在 include/uapi/linux/major.h 文件中，定义如下：

```
#define INPUT_MAJOR 13
```

因此，input 子系统的所有设备主设备号都为 13，我们在使用 input 子系统处理输入设备的时候就不需要去注册字符设备了，我们只需要向系统注册一个 input_device 即可。

1、注册 input_dev

在使用 input 子系统的时候我们只需要注册一个 input 设备即可, input_dev 结构体表示 input 设备, 此结构体定义在 include/linux/input.h 文件中, 定义如下(有省略):

示例代码 22.1.2.2 input_dev 结构体

```

1 struct input_dev {
2     const char *name;
3     const char *phys;
4     const char *uniq;
5     struct input_id id;
6
7     unsigned long propbit[BITS_TO_LONGS(INPUT_PROP_CNT)];
8
9     unsigned long evbit[BITS_TO_LONGS(EV_CNT)]; /* 事件类型的位图 */
10    unsigned long keybit[BITS_TO_LONGS(KEY_CNT)]; /* 按键值的位图 */
11    unsigned long relbit[BITS_TO_LONGS(REL_CNT)]; /* 相对坐标的位图 */
12    unsigned long absbit[BITS_TO_LONGS(ABS_CNT)]; /* 绝对坐标的位图 */
13    unsigned long msckbit[BITS_TO_LONGS(MSC_CNT)]; /* 杂项事件的位图 */
14    unsigned long ledbit[BITS_TO_LONGS(LED_CNT)]; /* LED 相关的位图 */
15    unsigned long sndbit[BITS_TO_LONGS(SND_CNT)]; /* sound 有关的位图 */
16    unsigned long ffbbit[BITS_TO_LONGS(FF_CNT)]; /* 压力反馈的位图 */
17    unsigned long swbit[BITS_TO_LONGS(SW_CNT)]; /* 开关状态的位图 */
18    .....
19    bool devres_managed;
20 };
    
```

第 9 行, evbit 表示输入事件类型, 可选的事件类型定义在 include/uapi/linux/input.h 文件中, 事件类型如下:

示例代码 22.1.2.3 事件类型

```

1 #define EV_SYN          0x00 /* 同步事件 */
2 #define EV_KEY          0x01 /* 按键事件 */
3 #define EV_REL          0x02 /* 相对坐标事件 */
4 #define EV_ABS          0x03 /* 绝对坐标事件 */
5 #define EV_MSC          0x04 /* 杂项(其他)事件 */
6 #define EV_SW           0x05 /* 开关事件 */
7 #define EV_LED          0x11 /* LED */
8 #define EV_SND          0x12 /* sound(声音) */
9 #define EV_REP          0x14 /* 重复事件 */
10 #define EV_FF           0x15 /* 压力事件 */
11 #define EV_PWR          0x16 /* 电源事件 */
12 #define EV_FF_STATUS    0x17 /* 压力状态事件 */
    
```

比如本章我们要使用到按键, 那么就需要注册 EV_KEY 事件, 如果要使用连接功能的话还需要注册 EV_REP 事件。

继续回到示例代码 22.1.2.2 中, 第 9 行~17 行的 evbit、keybit、relbit 等等都是存放不同事件对应的值。比如我们本章要使用按键事件, 因此要用到 keybit, keybit 就是按键事件使用的位

图, Linux 内核定义了很多按键值, 这些按键值定义在 `include/uapi/linux/input-event-codes.h` 文件中, 按键值如下:

```

示例代码 22.1.2.4 按键值
1  #define KEY_RESERVED      0
2  #define KEY_ESC           1
3  #define KEY_1             2
4  #define KEY_2             3
5  #define KEY_3             4
6  #define KEY_4             5
7  #define KEY_5             6
8  #define KEY_6             7
9  #define KEY_7             8
10 #define KEY_8             9
11 #define KEY_9            10
12 #define KEY_0            11
...
    
```

我们可以将开发板上的按键值设置为示例代码 22.1.2.4 中的任意一个, 本章我们依旧使用《第十三章 Linux 按键输入实验》里面的 GPIO3_C5 引脚来模拟按键, 然后将其按键值设置为 KEY_0。

在编写 input 设备驱动的时候我们需要先申请一个 `input_dev` 结构体变量, 使用 `input_allocate_device` 函数来申请一个 `input_dev`, 此函数原型如下所示:

```
struct input_dev *input_allocate_device(void)
```

函数参数和返回值含义如下:

参数: 无。

返回值: 申请到的 `input_dev`。

如果要注销 input 设备的话需要使用 `input_free_device` 函数来释放掉前面申请到的 `input_dev`, `input_free_device` 函数原型如下:

```
void input_free_device(struct input_dev *dev)
```

函数参数和返回值含义如下:

dev: 需要释放的 `input_dev`。

返回值: 无。

申请好一个 `input_dev` 以后就需要初始化这个 `input_dev`, 需要初始化的内容主要为事件类型(evbit)和事件值(keybit)这两种。`input_dev` 初始化完成以后就需要向 Linux 内核注册 `input_dev` 了, 需要用到 `input_register_device` 函数, 此函数原型如下:

```
int input_register_device(struct input_dev *dev)
```

函数参数和返回值含义如下:

dev: 要注册的 `input_dev`。

返回值: 0, `input_dev` 注册成功; 负值, `input_dev` 注册失败。

同样的, 注销 input 驱动的时候也需要使用 `input_unregister_device` 函数来注销掉前面注册的 `input_dev`, `input_unregister_device` 函数原型如下:

```
void input_unregister_device(struct input_dev *dev)
```

函数参数和返回值含义如下:

dev: 要注销的 `input_dev`。

返回值: 无。

综上所述, input_dev 注册过程如下:

- ①、使用 input_allocate_device 函数申请一个 input_dev。
- ②、初始化 input_dev 的事件类型以及事件值。
- ③、使用 input_register_device 函数向 Linux 系统注册前面初始化好的 input_dev。
- ④、卸载 input 驱动的时候需要先使用 input_unregister_device 函数注销掉注册的 input_dev, 然后使用 input_free_device 函数释放掉前面申请的 input_dev。input_dev 注册过程示例代码如下所示:

示例代码 22.1.2.5 input_dev 注册流程

```

1  struct input_dev *inputdev;      /* input 结构体变量 */
2
3  /* 驱动入口函数 */
4  static int __init xxx_init(void)
5  {
6      .....
7      inputdev = input_allocate_device(); /* 申请 input_dev */
8      inputdev->name = "test_inputdev"; /* 设置 input_dev 名字 */
9
10     /******第一种设置事件和事件值的方法******/
11     __set_bit(EV_KEY, inputdev->evbit); /* 设置产生按键事件 */
12     __set_bit(EV_REP, inputdev->evbit); /* 重复事件 */
13     __set_bit(KEY_0, inputdev->keybit); /* 设置产生哪些按键值 */
14     /*******/
15
16     /******第二种设置事件和事件值的方法******/
17     keyinputdev.inputdev->evbit[0] = BIT_MASK(EV_KEY) |
18                                     BIT_MASK(EV_REP);
19     keyinputdev.inputdev->keybit[BIT_WORD(KEY_0)] |=
20                                     BIT_MASK(KEY_0);
21     /*******/
22     /******第三种设置事件和事件值的方法******/
23     keyinputdev.inputdev->evbit[0] = BIT_MASK(EV_KEY) |
24                                     BIT_MASK(EV_REP);
25     input_set_capability(keyinputdev.inputdev, EV_KEY, KEY_0);
26     /*******/
27     /* 注册 input_dev */
28     input_register_device(inputdev);
29     .....
30     return 0;
31 }
    
```

```

32 /* 驱动出口函数 */
33 static void __exit xxx_exit(void)
34 {
35     input_unregister_device(inputdev);    /* 注销 input_dev */
36     input_free_device(inputdev);         /* 删除 input_dev */
37 }
    
```

第 1 行，定义一个 `input_dev` 结构体指针变量。

第 4~30 行，驱动入口函数，在此函数中完成 `input_dev` 的申请、设置、注册等工作。第 7 行调用 `input_allocate_device` 函数申请一个 `input_dev`。第 10~23 行都是设置 `input` 设备事件和按键值，这里用了三种方法来设置事件和按键值。第 27 行调用 `input_register_device` 函数向 Linux 内核注册 `inputdev`。

第 33~37 行，驱动出口函数，第 35 行调用 `input_unregister_device` 函数注销前面注册的 `input_dev`，第 36 行调用 `input_free_device` 函数删除前面申请的 `input_dev`。

2、上报输入事件

当我们向 Linux 内核注册好 `input_dev` 以后还不能高枕无忧的使用 `input` 设备，`input` 设备都是具有输入功能的，但是具体是什么样的输入值 Linux 内核是不知道的，我们需要获取到具体的输入值，或者说是输入事件，然后将输入事件上报给 Linux 内核。比如按键，我们需要在按键中断处理函数，或者消抖定时器中断函数中将按键值上报给 Linux 内核，这样 Linux 内核才能获得到正确的输入值。不同的事件，其上报事件的 API 函数不同，我们依次来看一下一些常用的事件上报 API 函数。

首先是 `input_event` 函数，此函数用于上报指定的事件以及对应的值，函数原型如下：

```

void input_event(struct input_dev *dev,
                 unsigned int type,
                 unsigned int code,
                 int value)
    
```

函数参数和返回值含义如下：

dev: 需要上报的 `input_dev`。

type: 上报的事件类型，比如 `EV_KEY`。

code: 事件码，也就是我们注册的按键值，比如 `KEY_0`、`KEY_1` 等等。

value: 事件值，比如 1 表示按键按下，0 表示按键松开。

返回值: 无。

`input_event` 函数可以上报所有的事件类型和事件值，Linux 内核也提供了其他的针对具体事件的上报函数，这些函数其实都用到了 `input_event` 函数。比如上报按键所使用的 `input_report_key` 函数，此函数内容如下：

```

// 例代码 22.1.2.6 input_report_key 函数
static inline void input_report_key(struct input_dev *dev,
                                   unsigned int code, int value)
{
    input_event(dev, EV_KEY, code, !!value);
}
    
```

从示例代码 22.1.2.6 可以看出，`input_report_key` 函数的本质就是 `input_event` 函数，如果要上报按键事件的话还是建议大家使用 `input_report_key` 函数。

同样的还有一些其他的事件上报函数，这些函数如下所示：

```

void input_report_rel(struct input_dev *dev, unsigned int code, int value)
void input_report_abs(struct input_dev *dev, unsigned int code, int value)
void input_report_ff_status(struct input_dev *dev, unsigned int code, int value)
void input_report_switch(struct input_dev *dev, unsigned int code, int value)
void input_mt_sync(struct input_dev *dev)
    
```

当我们上报事件以后还需要使用 `input_sync` 函数来告诉 Linux 内核 input 子系统上报结束, `input_sync` 函数本质是上报一个同步事件, 此函数原型如下所示:

```
void input_sync(struct input_dev *dev)
```

函数参数和返回值含义如下:

dev: 需要上报同步事件的 `input_dev`。

返回值: 无。

综上所述, 按键的上报事件的参考代码如下所示:

示例代码 22.1.2.7 事件上报参考代码

```

1  /* 用于按键消抖的定时器服务函数 */
2  void timer_function(unsigned long arg)
3  {
4      unsigned char value;
5
6      value = gpio_get_value(keydesc->gpio);    /* 读取 IO 值    */
7      if(value == 0){                            /* 按下按键    */
8          /* 上报按键值 */
9          input_report_key(inputdev, KEY_0, 1); /* 最后一个参数 1, 按下 */
10         input_sync(inputdev);                /* 同步事件    */
11     } else {                                    /* 按键松开    */
12         input_report_key(inputdev, KEY_0, 0); /* 最后一个参数 0, 松开 */
13         input_sync(inputdev);                /* 同步事件    */
14     }
15 }
    
```

第 6 行, 获取按键值, 判断按键是否按下。

第 9~10 行, 如果按键值为 0 那么表示按键被按下了, 如果按键按下的话就要使用 `input_report_key` 函数向 Linux 系统上报按键值, 比如向 Linux 系统通知 `KEY_0` 这个按键按下了。

第 12~13 行, 如果按键值为 1 的话就表示按键没有按下, 是松开的。向 Linux 系统通知 `KEY_0` 这个按键没有按下或松开了。

22.1.3 input_event 结构体

Linux 内核使用 `input_event` 这个结构体来表示所有的输入事件, `input_event` 结构体定义在 `include/uapi/linux/input.h` 文件中, 结构体内容如下:

示例代码 22.1.3.1 input_event 结构体

```

1 struct input_event {
2     struct timeval time;
3     __u16 type;
4     __u16 code;
    
```

```
5  __s32 value;
6  };
```

我们依次来看一下 `input_event` 结构体中的各个成员变量:

time: 时间, 也就是此事件发生的时间, 为 `timeval` 结构体类型, `timeval` 结构体定义如下:

示例代码 22.1.3.2 `timeval` 结构体

```
1 typedef long          __kernel_long_t;
2 typedef __kernel_long_t __kernel_time_t;
3 typedef __kernel_long_t __kernel_suseconds_t;
4
5 struct timeval {
6     __kernel_time_t      tv_sec;    /* 秒 */
7     __kernel_suseconds_t tv_usec;  /* 微秒 */
8 };
```

从示例代码 22.1.3.2 可以看出, `tv_sec` 和 `tv_usec` 这两个成员变量都为 `long` 类型, 也就是 32 位, 这个一定要记住, 后面我们分析 `event` 事件上报数据的时候要用到。

type: 事件类型, 比如 `EV_KEY`, 表示此次事件为按键事件, 此成员变量为 16 位。

code: 事件码, 比如在 `EV_KEY` 事件中 `code` 就表示具体的按键码, 如: `KEY_0`、`KEY_1` 等等这些按键。此成员变量为 16 位。

value: 值, 比如 `EV_KEY` 事件中 `value` 就是按键值, 表示按键有没有被按下, 如果为 1 的话说明按键按下, 如果为 0 的话说明按键没有被按下或者按键松开了。

`input_event` 这个结构体非常重要, 因为所有的输入设备最终都是按照 `input_event` 结构体呈现给用户的, 用户应用程序可以通过 `input_event` 来获取到具体的输入事件或相关的值, 比如按键值等。关于 `input` 子系统就讲解到这里, 接下来我们就以开发板上的 `GPIO3_C5` 这个引脚为例, 讲解一下如何编写 `input` 驱动。

22.2 硬件原理图分析

本章实验硬件原理图参考 13.2 小节即可。本章我们依旧使用《第十三章 Linux 按键输入实验》里面的 `GPIO3_C5` 引脚来模拟按键。

22.3 实验程序编写

本实验对应的例程路径为: [开发板光盘](#) → [01](#)、[程序源码](#) → [Linux 驱动例程](#) → [19_input](#)。

22.3.1 修改设备树文件

直接使用 15.3.1 小节中创建的 `key` 节点即可。

22.3.2 按键 `input` 驱动程序编写

新建名为“19_input”的文件夹, 然后在 `19_input` 文件夹里面创建 `vscode` 工程, 工作区命名为“`keyinput`”。工程创建好以后新建 `keyinput.c` 文件, 在 `keyinput.c` 里面输入如下内容:

示例代码 22.3.2.1 `keyinput.c` 文件代码段

```
1 #include <linux/module.h>
2 #include <linux/errno.h>
3 #include <linux/of.h>
```

```

4  #include <linux/platform_device.h>
5  #include <linux/of_gpio.h>
6  #include <linux/input.h>
7  #include <linux/timer.h>
8  #include <linux/of_irq.h>
9  #include <linux/interrupt.h>
10
11 #define KEYINPUT_NAME      "keyinput" /* 名字      */
12
13 /* key 设备结构体 */
14 struct key_dev{
15     struct input_dev *idev; /* 按键对应的 input_dev 指针 */
16     struct timer_list timer; /* 消抖定时器 */
17     int gpio_key;          /* 按键对应的 GPIO 编号 */
18     int irq_key;          /* 按键对应的中断号 */
19 };
20
21 static struct key_dev key; /* 按键设备 */
22
23 /*
24  * @description   : 按键中断服务函数
25  * @param - irq   : 触发该中断事件对应的中断号
26  * @param - arg   : arg 参数可以在申请中断的时候进行配置
27  * @return        : 中断执行结果
28  */
29 static irqreturn_t key_interrupt(int irq, void *dev_id)
30 {
31     if(key.irq_key != irq)
32         return IRQ_NONE;
33
34     /* 按键防抖处理, 开启定时器延时 15ms */
35     disable_irq_nosync(irq); /* 禁止按键中断 */
36     mod_timer(&key.timer, jiffies + msecs_to_jiffies(15));
37
38     return IRQ_HANDLED;
39 }
40
41 /*
42  * @description   : 按键初始化函数
43  * @param - nd     : device_node 设备指针
44  * @return        : 成功返回 0, 失败返回负数
45  */
46 static int key_gpio_init(struct device_node *nd)
    
```

```

47 {
48     int ret;
49     unsigned long irq_flags;
50
51     /* 从设备树中获取 GPIO */
52     key.gpio_key = of_get_named_gpio(nd, "key-gpio", 0);
53     if(!gpio_is_valid(key.gpio_key)) {
54         printk("key: Failed to get key-gpio\n");
55         return -EINVAL;
56     }
57
58     /* 申请使用 GPIO */
59     ret = gpio_request(key.gpio_key, "KEY0");
60     if (ret) {
61         printk(KERN_ERR "key: Failed to request key-gpio\n");
62         return ret;
63     }
64
65     /* 将 GPIO 设置为输入模式 */
66     gpio_direction_input(key.gpio_key);
67
68     /* 获取 GPIO 对应的中断号 */
69     key.irq_key = irq_of_parse_and_map(nd, 0);
70     if(!key.irq_key){
71         return -EINVAL;
72     }
73
74     /* 获取设备树中指定的中断触发类型 */
75     irq_flags = irq_get_trigger_type(key.irq_key);
76     if (IRQF_TRIGGER_NONE == irq_flags)
77         irq_flags = IRQF_TRIGGER_FALLING | IRQF_TRIGGER_RISING;
78
79     /* 申请中断 */
80     ret = request_irq(key.irq_key, key_interrupt, irq_flags,
81                     "Key0_IRQ", NULL);
82     if (ret) {
83         gpio_free(key.gpio_key);
84         return ret;
85     }
86     return 0;
87 }
88
    
```

```

89  /*
90  * @description   : 定时器服务函数, 用于按键消抖, 定时时间到了以后
91  *                : 再读取按键值, 根据按键的状态上报相应的事件
92  * @param - arg   : arg 参数就是定时器的结构体
93  * @return        : 无
94  */
95  static void key_timer_function(struct timer_list *arg)
96  {
97      int val;
98
99      /* 读取按键值并上报按键事件 */
100     val = gpio_get_value(key.gpio_key);
101     input_report_key(key.idev, KEY_0, !val);
102     input_sync(key.idev);
103
104     enable_irq(key.irq_key);
105 }
106
107 /*
108 * @description   : platform 驱动的 probe 函数, 当驱动与设备匹配成功
109 *                : 以后此函数会被执行
110 * @param - pdev  : platform 设备指针
111 * @return        : 0, 成功; 其他负值, 失败
112 */
113 static int atk_key_probe(struct platform_device *pdev)
114 {
115     int ret;
116
117     /* 初始化 GPIO */
118     ret = key_gpio_init(pdev->dev.of_node);
119     if(ret < 0)
120         return ret;
121
122     /* 初始化定时器 */
123     timer_setup(&key.timer, key_timer_function, 0);
124
125     /* 申请 input_dev */
126     key.idev = input_allocate_device();
127     key.idev->name = KEYINPUT_NAME;
128
129     #if 0
130     /* 初始化 input_dev, 设置产生哪些事件 */
131     __set_bit(EV_KEY, key.idev->evbit); /* 设置产生按键事件 */
    
```

```

132     __set_bit(EV_REP, key.idev->evbit); /* 重复事件, 比如按下去不放开, 就
                                           会一直输出信息 */
133
134     /* 初始化 input_dev, 设置产生哪些按键 */
135     __set_bit(KEY_0, key.idev->keybit);
136 #endif
137
138 #if 0
139     key.idev->evbit[0] = BIT_MASK(EV_KEY) | BIT_MASK(EV_REP);
140     key.idev->keybit[BIT_WORD(KEY_0)] |= BIT_MASK(KEY_0);
141 #endif
142
143     key.idev->evbit[0] = BIT_MASK(EV_KEY) | BIT_MASK(EV_REP);
144     input_set_capability(key.idev, EV_KEY, KEY_0);
145
146     /* 注册输入设备 */
147     ret = input_register_device(key.idev);
148     if (ret) {
149         printk("register input device failed!\r\n");
150         goto free_gpio;
151     }
152
153     return 0;
154 free_gpio:
155     free_irq(key.irq_key, NULL);
156     gpio_free(key.gpio_key);
157     del_timer_sync(&key.timer);
158     return -EIO;
159
160 }
161
162 /*
163  * @description   : platform 驱动的 remove 函数, 当 platform 驱动模块
164  *                 卸载时此函数会被执行
165  * @param - dev   : platform 设备指针
166  * @return        : 0, 成功;其他负值, 失败
167  */
168 static int atk_key_remove(struct platform_device *pdev)
169 {
170     free_irq(key.irq_key, NULL); /* 释放中断号 */
171     gpio_free(key.gpio_key);    /* 释放 GPIO */
172     del_timer_sync(&key.timer); /* 删除 timer */
173     input_unregister_device(key.idev); /* 释放 input_dev */

```

```

174
175     return 0;
176 }
177
178 static const struct of_device_id key_of_match[] = {
179     {.compatible = "alientek,key"},
180     /* Sentinel */
181 };
182
183 static struct platform_driver atk_key_driver = {
184     .driver = {
185         .name = "rk3568-key",
186         .of_match_table = key_of_match,
187     },
188     .probe = atk_key_probe,
189     .remove = atk_key_remove,
190 };
191
192 module_platform_driver(atk_key_driver);
193
194 MODULE_LICENSE("GPL");
195 MODULE_AUTHOR("ALIENTEK");
196 MODULE_INFO(intree, "Y");
    
```

在本程序中用到了定时器和中断相关的 API 函数，这些内容在前面章节都已经给大家介绍过了，而本章的重点知识点是 linux 下的 input 子系统，那下面我们将对 keyinput.c 代码进行讲解。

第 14~19 行，自定义的按键设备结构体 `struct key_dev`，用于描述一个按键设备，其中的成员变量包括一个 `input_dev` 指针变量，定时器 `timer`、GPIO 以及中断号。

第 29~39 行，按键中断处理函数 `key_interrupt`，当按键按下或松开的时候都会触发，也就是上升沿和下降沿都会触发此中断。`key_interrupt` 函数中的操作也很简单，调用 `disable_irq_nosync` 函数先禁止中断，然后使用 `mod_timer` 打开定时器，定时时长为 15ms。

第 46~87 行，按键的 GPIO 初始化。

第 95~105 行，定时器服务函数 `key_timer_function`，使用到定时器的目的主要是为了使用软件的方式进行按键消抖处理，在 `key_timer_function` 函数中，我们使用 `gpio_get_value` 获取按键 GPIO 的电平状态，使用 `input_report_key` 函数上报按键事件。按键按下以后 `val` 为 1，释放以后 `val` 为 0。事件上报完成之后使用 `input_sync` 函数同步事件，表示此事件已上报完成，input 子系统核心层就会进行相关的处理。第 104 行 `enable_irq` 函数使能中断，因为在按键中断发生的时候我们会关闭中断，等事件处理完成之后再打开。

第 113~160 行，platform 驱动的 probe 函数 `atk_key_probe`，其中第 136~161 行，使用 `input_allocate_device` 函数申请 `input_dev`，然后设置相应的事件以及事件码(也就是 KEY 模拟成那个按键，这里我们设置为 `KEY_0`)。最后使用 `input_register_device` 函数向 Linux 内核注册 `input_dev`。

第 168~176 行，platform 驱动的 remove 函数 `mykey_remove`，在该函数中先释放 GPIO 在

使用 `del_timer_sync` 删除定时器并且调用 `input_unregister_device` 卸载按键设备。

22.3.3 编写测试 APP

新建 `keyinputApp.c` 文件，然后在里面输入如下所示内容：

示例代码 22.3.3.1 `keyinputApp.c` 文件代码段

```

1  #include <stdio.h>
2  #include <unistd.h>
3  #include <sys/types.h>
4  #include <sys/stat.h>
5  #include <fcntl.h>
6  #include <stdlib.h>
7  #include <string.h>
8  #include <linux/input.h>
9
10 /*
11  * @description   : main 主程序
12  * @param - argc  : argv 数组元素个数
13  * @param - argv  : 具体参数
14  * @return        : 0 成功;其他 失败
15  */
16 int main(int argc, char *argv[])
17 {
18     int fd, ret;
19     struct input_event ev;
20
21     if(2 != argc) {
22         printf("Usage:\n"
23             "\t./keyinputApp /dev/input/eventX   @ Open Key\n"
24             );
25         return -1;
26     }
27
28     /* 打开设备 */
29     fd = open(argv[1], O_RDWR);
30     if(0 > fd) {
31         printf("Error: file %s open failed!\r\n", argv[1]);
32         return -1;
33     }
34
35     /* 读取按键数据 */
36     for ( ; ; ) {
37
38         ret = read(fd, &ev, sizeof(struct input_event));
    
```

```

39     if (ret) {
40         switch (ev.type) {
41             case EV_KEY:                /* 按键事件          */
42                 if (KEY_0 == ev.code) { /* 判断是不是 KEY_0 按键 */
43                     if (ev.value)      /* 按键按下          */
44                         printf("Key0 Press\n");
45                     else                /* 按键松开          */
46                         printf("Key0 Release\n");
47                 }
48                 break;
49
50                 /* 其他类型的事件, 自行处理 */
51             case EV_REL:
52                 break;
53             case EV_ABS:
54                 break;
55             case EV_MSC:
56                 break;
57             case EV_SW:
58                 break;
59         };
60     }
61     else {
62         printf("Error: file %s read failed!\r\n", argv[1]);
63         goto out;
64     }
65 }
66
67 out:
68     /* 关闭设备 */
69     close(fd);
70     return 0;
71 }
    
```

第 22.1.3 小节已经说过了, Linux 内核会使用 `input_event` 结构体来表示输入事件, 所以我们要获取按键输入信息, 那么必须借助于 `input_event` 结构体。第 19 行定义了一个 `input_event` 类型变量 `ev`。

第 36~65 行, 当我们向 Linux 内核成功注册 `input_dev` 设备以后, 会在 `/dev/input` 目录下生成一个名为“`eventX(X=0...n)`”的文件, 这个 `/dev/input/eventX` 就是对应的 `input` 设备文件。我们读取这个文件就可以获取到输入事件信息, 比如按键值什么的。使用 `read` 函数读取输入设备文件, 也就是 `/dev/input/eventX`, 读取到的数据按照 `input_event` 结构体组织起来。获取到输入事件以后(`input_event` 结构体类型)使用 `switch case` 语句来判断事件类型, 本章实验我们设置的事件类型为 `EV_KEY`, 因此只需要处理 `EV_KEY` 事件即可。比如获取按键编号(`KEY_0` 的编号为 11)、获取按键状态, 按下还是松开的?

22.4 运行测试

22.4.1 编译驱动程序和测试 APP

1、编译驱动程序

编写 Makefile 文件，本章实验的 Makefile 文件和第五章实验基本一样，只是将 obj-m 变量的值改为“miscled.o”，Makefile 内容如下所示：

示例代码 22.4.1.1 Makefile 文件

```
1 KERNELDIR := /home/alientek/rk3568_linux_sdk/kernel
.....
4 obj-m := keyinput.o
.....
11 clean:
12 $(MAKE) -C $(KERNELDIR) M=$(CURRENT_PATH) clean
```

第 4 行，设置 obj-m 变量的值为“keyinput.o”。

输入如下命令编译出驱动模块文件：

```
make ARCH=arm64 //ARCH=arm64 必须指定，否则编译会失败
```

编译成功以后就会生成一个名为“keyinput.ko”的驱动模块文件。

2、编译测试 APP

输入如下命令编译测试 keyinputApp.c 这个测试程序：

```
/opt/atk-dlrk356x-toolchain/bin/aarch64-buildroot-linux-gnu-gcc keyinputApp.c -o keyinputApp
```

编译成功以后就会生成 keyinputApp 这个应用程序。

22.4.2 运行测试

在 Ubuntu 中将上一小节编译出来的 keyinput.ko 和 keyinputApp 通过 adb 命令发送到开发板的/lib/modules/4.19.232 目录下，命令如下：

```
adb push keyinput.ko keyinputApp /lib/modules/4.19.232
```

重启开发板，进入到目录 lib/modules/5.4.31 中。在加载 keyinput.ko 驱动模块之前，先看一下/dev/input 目录下都有哪些文件，结果如图 22.4.2.1 所示：

```
root@ATK-DLRK356X:/lib/modules/4.19.232# ls /dev/input
by-path event1 event3 event5 event7 event9
event0 event2 event4 event6 event8
root@ATK-DLRK356X:/lib/modules/4.19.232#
```

图 22.4.2.1 /dev/input 目录

从图 22.4.2.1 可以看出，默认出厂系统中/dev/input 目录下有 event0~2 三个文件。

接下来输入如下命令加载 keyinput.ko 这个驱动模块：

```
depmod //第一次加载驱动的时候需要运行此命令
```

```
modprobe keyinput //加载驱动模块
```

当驱动模块加载成功以后再来看一下/dev/input 目录下有哪些文件，结果如图 22.4.2.2 所示：

```
root@ATK-DLRK356X:/lib/modules/4.19.232# ls /dev/input
by-path event1 event2 event4 event6 event8
event0 event10 event3 event5 event7 event9
root@ATK-DLRK356X:/lib/modules/4.19.232#
```

本实验对应的event10

图 22.4.2.2 加载驱动以后的/dev/input 目录

从图 22.4.2.2 可以看出，加载驱动以后在/dev/input 目录下生成了一个 event10 文件，这其实就是我们注册的驱动所对应的设备文件。keyinputApp 就是通过读取/dev/input/event10 这个文件来获取输入事件信息的，输入如下测试命令：

```
./keyinputApp /dev/input/event10
```

使用杜邦线将图 13.2.1 中 GPIO3_C5 这个 IO 接到开发板的 3.3V 电压上并拔出，模拟按键被按下与释放过程，结果如图 22.4.2.3 所示：

```
root@ATK-DLRK356X:/lib/modules/4.19.232# ./keyinputApp /dev/input/event10
Key0 Press
Key0 Release
Key0 Press
Key0 Release
Key0 Press
Key0 Release
Key0 Press
Key0 Release
```

图 22.4.2.3 测试结果

从图 22.4.2.3 可以看出，当我们按下或者释放开发板上的按键以后都会在终端上输出相应的内容，提示我们哪个按键按下或释放了，在 Linux 内核中 KEY_0 为 11。

另外，我们也可以不用 keyinputApp 来测试驱动，可以直接使用 hexdump 命令来查看/dev/input/event10 文件内容，输入如下命令：

```
hexdump /dev/input/event10
```

同样模拟按键按下，输出信息如下图所示：

```
root@ATK-DLRK356X:/lib/modules/4.19.232# hexdump /dev/input/event10
00000000 d61b 647e 0000 0000 5e39 000a 0000 0000
00000010 0001 000b 0001 0000 d61b 647e 0000 0000
00000020 5e39 000a 0000 0000 0000 0000 0000 0000
00000030 d61b 647e 0000 0000 892e 000e 0000 0000
00000040 0001 000b 0002 0000 d61b 647e 0000 0000
00000050 892e 000e 0000 0000 0000 0000 0001 0000
00000060 d61b 647e 0000 0000 1866 000f 0000 0000
00000070 0001 000b 0002 0000 d61b 647e 0000 0000
00000080 1866 000f 0000 0000 0000 0000 0001 0000
```

图 22.4.2.4 原始数据值

图 22.4.2.4 就是 input_event 类型的原始事件数据值，采用十六进制表示，这些原始数据的含义如下：

```
示例代码 22.4.2.1 input_event 类型的原始事件值
/*****input_event 类型*****/
/* 编号 */ /* tv_sec */ /* tv_usec */ /* type */ /* code */ /* value */
00000000 5378 6445 228b 000b 0001 000b 0001 0000
00000010 5378 6445 228b 000b 0000 0000 0000 0000
00000020 5378 6445 d913 000e 0001 000b 0000 0000
00000030 5378 6445 d913 000e 0000 0000 0000 0000
```

type 为事件类型，查看示例代码 22.1.2.3 可知，EV_KEY 事件值为 1，EV_SYN 事件值为 0。因此第 1 行表示 EV_KEY 事件，第 2 行表示 EV_SYN 事件。code 为事件编码，也就是按键号，查看示例代码 22.1.2.4 可以，KEY_0 这个按键编号为 11，对应的十六进制为 0xb，因此第 1 行表示 KEY_0 这个按键事件，最后的 value 就是按键值，为 1 表示按下，为 0 的话表示松开。综上所述，示例代码 22.4.2.1 中的原始事件值含义如下：

第 1 行，按键(KEY_0)按下事件。

第 2 行, EV_SYN 同步事件, 因为每次上报按键事件以后都要同步的上报一个 EV_SYN 事件。

第 3 行, 按键(KEY_0)松开事件。

第 4 行, EV_SYN 同步事件, 和第 2 行一样。

22.5 Linux 自带按键驱动程序的使用

22.5.1 使能内核自带按键驱动程序源码简析

一般我们直接用 GPIO 引脚来驱动按键, 但是这样会浪费大量的 IO。瑞芯微官方使用 ADC 来驱动按键, 不同的按键按下以后 ADC 值就不同, 这样就可以通过采集具体电压来区分哪个按键按下了。这样就可以通过一个 ADC 引脚来外接很多按键, 省下了宝贵的 GPIO 资源。

Linux 主线内核自带的 KEY 驱动默认是用 GPIO 来连接按键的, 如果你的产品使用 GPIO 来直接驱动按键, 那么就可以使用此驱动。按照如下路径找到相应的配置选项:

- Device Drivers
 - Input device support
 - Generic input layer (needed for keyboard, mouse, ...) (INPUT [=y])
 - Keyboards (INPUT_KEYBOARD [=y])
 - GPIO Buttons

选中“GPIO Buttons”选项, 将其编译进 Linux 内核中, 如图 22.5.1.1 所示:

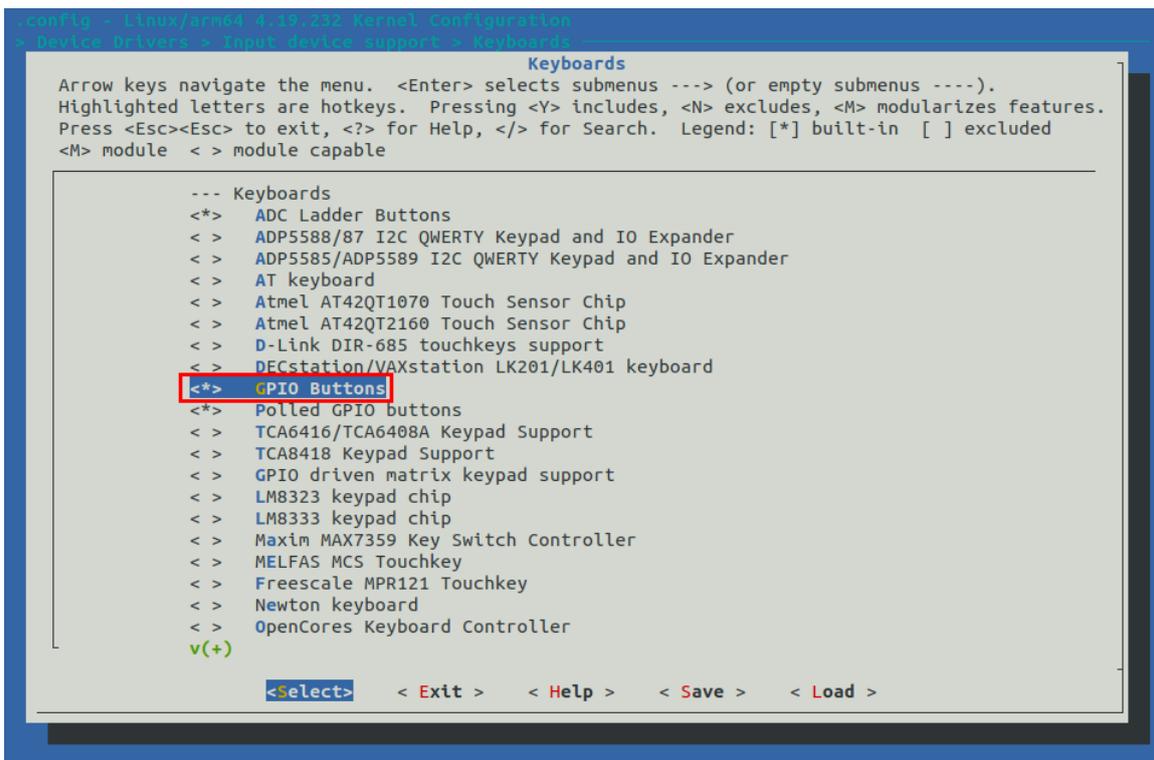


图 22.5.1.1 内核自带 KEY 驱动使能选项

前面说了, 瑞芯微官方使用 ADC 来驱动按键, 所以图 22.5.1.1 中的通用按键驱动本节就用不了。应该使用“ADC Ladder Buttons”驱动, 配置路径如下:

- Device Drivers
 - Input device support

- Generic input layer (needed for keyboard, mouse, ...) (INPUT [=y])
 - Keyboards (INPUT_KEYBOARD [=y])
 - ADC Ladder Buttons

选中以后如图 22.5.1.2 所示:

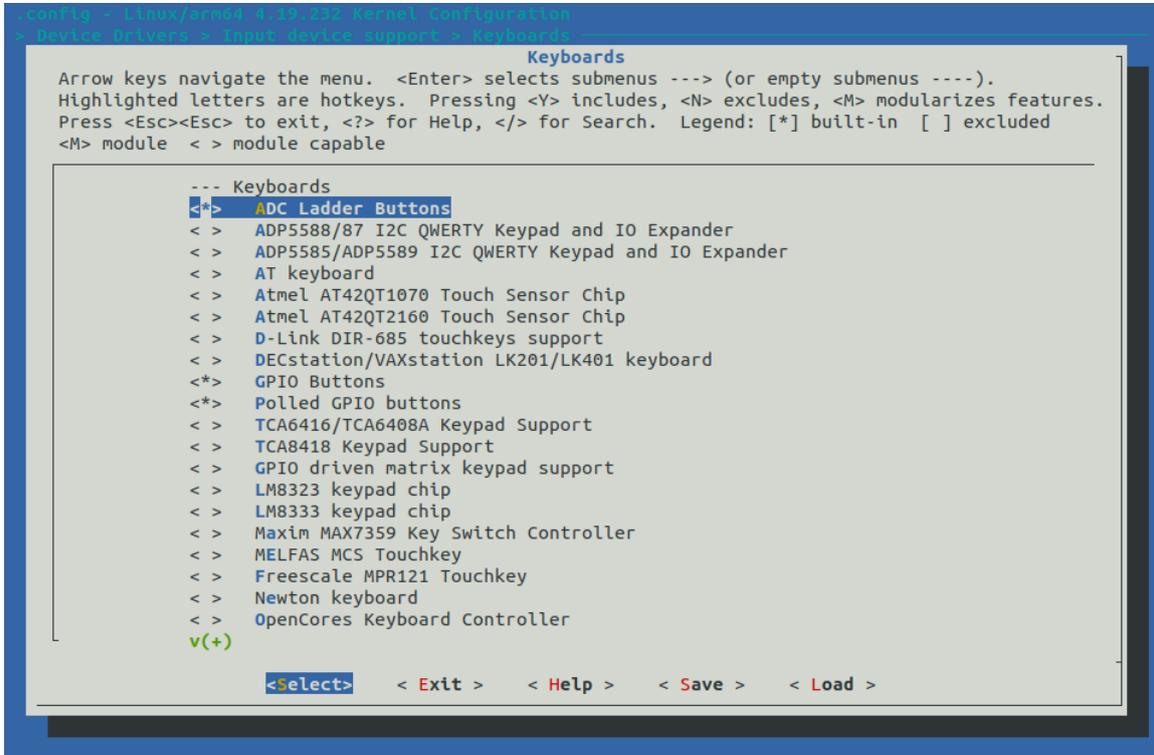


图 22.5.1.2 使能 ADC 按键驱动

对应的驱动文件为 drivers/input/keyboard/adc-keys.c。

22.5.2 自带 ADC 按键驱动程序的使用

要使用 Linux 内核自带的 ADC 按键驱动程序很简单，只需要根据 Documentation/devicetree/bindings/input/adc-keys.txt 这个文件在设备树中添加指定的设备节点即可，节点要求如下：

- ①、节点名字为“adc-keys”。
- ②、adc-keys 节点的 compatible 属性值一定要设置为“adc-keys”。
- ③、io-channel 属性描述用于驱动按键的 ADC 通道。
- ④、io-channel-names 属性要设置为“buttons”。

⑤、keyup-threshold-microvolt 属性指定按键抬起时的电压，单位是微伏，正点原子 ATK-CLRK3568F 核心板上按键对应的 ADC 默认接到 1.8V 电压，所以按键抬起时就是 1.8V。keyup-threshold-microvolt 设置为 1800000。

另外也有两个可选的属性：

- ①、poll-interval: 轮询间隔，单位为 ms，也就是 ADC 多久轮询查看一次按键的 ADC 值。
- ②、autorepeat 属性描述是否支持连接，bool 类型，可以设置为“true”或“false”。

所有的 KEY 都是 adc-keys 的子节点，每个子节点可以用如下属性描述自己：

label: 按键名字。

linux,code: KEY 要模拟的按键，也就是示例代码 22.1.2.4 中的这些按键。

press-threshold-microvolt: 按键按下以后对应的 ADC 值, 单位为微伏。

这里我们将开发板上的四个用户按键都用起来, KEY4、KEY5、KEY6 和 KEY7 分别模拟为键盘上的: ESC、RIGHT、LEFT 和 MENU 按键。

打开 rk3568-atk-evb1-ddr4-v10.dtsi 文件, 我们出厂系统已经设置好了 adc-keys 节点, 内容如下:

示例代码 22.5.2.1 gpio-keys 节点内容

```

1  &adc_keys {
2      vol-up-key {
3          label = "volume up";
4          linux,code = <KEY_VOLUMEUP>;
5          press-threshold-microvolt = <17822>;
6      };
7
8      vol-down-key {
9          label = "volume down";
10         linux,code = <KEY_VOLUMEDOWN>;
11         press-threshold-microvolt = <415385>;
12     };
13
14     menu-key {
15         label = "menu";
16         linux,code = <KEY_MENU>;
17         press-threshold-microvolt = <805525>;
18     };
19
20     esc-key {
21         label = "esc";
22         linux,code = <KEY_ESC>;
23         press-threshold-microvolt = <1201993>;
24     };
25
26     /delete-node/ back-key;
27 };
    
```

第 2~6 行, V+ 按键子节点, 也就是 KEY4 按键, 此按键按下以后电压为 17822uV。

第 8~12 行, V- 按键子节点, 也就是 KEY5 按键, 此按键按下以后电压为 415385uV。

第 14~18 行, MENU 按键子节点, 也就是 KEY6 按键, 此按键按下以后电压为 805525uV。

第 20~24 行, ESC 按键子节点, 也就是 KEY7 按键, 此按键按下以后电压为 1201993uV。

系统启动以后查看 /dev/input 目录, 看看都有哪些文件, 结果如图 22.5.2.1 所示:

```

root@ATK-DLRK356X:/# ls /dev/input -l
total 0
drwxr-xr-x 2 root root    200 Jun  6 15:21 by-path
crw-rw---- 1 root input 13, 64 Jun  6 15:21 event0
crw-rw---- 1 root input 13, 65 Jun  6 15:21 event1
crw-rw---- 1 root input 13, 66 Jun  6 15:21 event2
crw-rw---- 1 root input 13, 67 Jun  6 15:21 event3
crw-rw---- 1 root input 13, 68 Jun  6 15:21 event4
crw-rw---- 1 root input 13, 69 Jun  6 15:21 event5
crw-rw---- 1 root input 13, 70 Jun  6 15:21 event6
crw-rw---- 1 root input 13, 71 Jun  6 15:21 event7
crw-rw---- 1 root input 13, 72 Jun  6 15:21 event8
crw-rw---- 1 root input 13, 73 Jun  6 15:21 event9
root@ATK-DLRK356X:/#
    
```

图 22.5.2.1 /dev/input 目录文件

从图 22.5.2.1 可以看出存在 event0~event9 这 10 个文件，其中肯定就有按键对应的文件。每个都测试一遍，就知道哪个文件是按键的了。这里是 event7 这个文件，输入如下命令：

```
hexdump /dev/input/event7
```

然后按下 ATK-DLRK3568 开发板上的 V+(KEY4)、ESC(KEY7)、V-(KEY5)和 MENU(KEY6)这四个按键，终端输出图 22.5.2.2 所示内容：

```

root@ATK-DLRK356X:/# hexdump /dev/input/event7
00000000 dfe7 647e 0000 0000 8bb3 0009 0000 0000
00000100 0001 0072 0001 0000 dfe7 647e 0000 0000
00000200 8bb3 0009 0000 0000 0000 0000 0000 0000
00000300 dfe7 647e 0000 0000 1e47 000b 0000 0000
00000400 0001 0072 0000 0000 dfe7 647e 0000 0000
00000500 1e47 000b 0000 0000 0000 0000 0000 0000
00000600 dfe8 647e 0000 0000 2a91 0002 0000 0000
00000700 0001 0073 0001 0000 dfe8 647e 0000 0000
00000800 2a91 0002 0000 0000 0000 0000 0000 0000
00000900 dfe8 647e 0000 0000 be42 0003 0000 0000
00000a00 0001 0073 0000 0000 dfe8 647e 0000 0000
00000b00 be42 0003 0000 0000 0000 0000 0000 0000
00000c00 dfe8 647e 0000 0000 794e 0008 0000 0000
00000d00 0001 008b 0001 0000 dfe8 647e 0000 0000
    
```

图 22.5.2.2 按键信息

如果按下按键以后会在终端上输出图 22.5.2.2 所示的信息那么就表示 Linux 内核的按键驱动工作正常。至于图 22.5.2.2 中内容的含义大家就自行分析，这个已经在 22.4.2 小节详细的分析过了，这里就不再讲解了。

第二十三章 Linux PWM 驱动实验

PWM 是很常用到功能，我们可以通过 PWM 来控制电机速度，也可以使用 PWM 来控制 LCD 的背光亮度。本章我们就来学习一下如何在 Linux 下进行 PWM 驱动开发。

23.1 PWM 驱动简析

PWM 全称是 Pulse Width Modulation，也就是脉冲宽度调制，PWM 信号如图 23.1.1 所示：

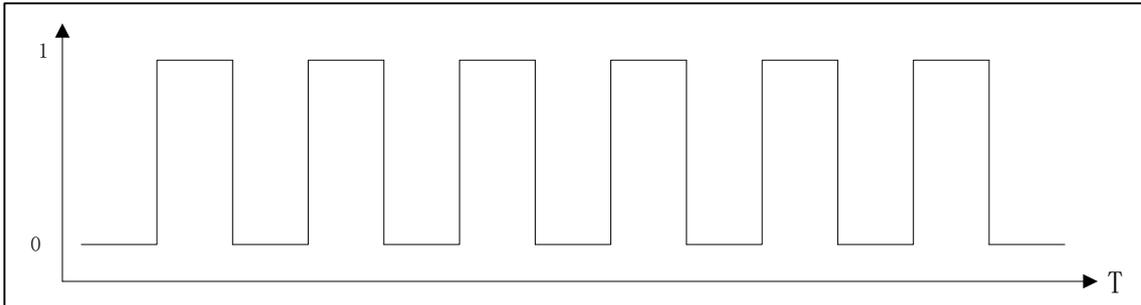


图 23.1.1 PWM 信号

PWM 信号有两个关键的术语：频率和占空比，频率就是开关速度，把一次开关算作一个周期，那么频率就是 1 秒内进行了多少次开关。占空比就是一个周期内高电平时间和低电平时间的比例，一个周期内高电平时间越长占空比就越大，反之占空比就越小。占空比用百分之表示，如果一个周期内全是低电平那么占空比就是 0%，如果一个周期内全是高电平那么占空比就是 100%。

我们给 LCD 的背光引脚输入一个 PWM 信号，这样就可以通过调整占空比的方式来调整 LCD 背光亮度了。提高占空比就会提高背光亮度，降低占空比就会降低背光亮度，重点就在于 PWM 信号的产生和占空比的控制。

23.1.1 设备树下的 PWM 控制器节点

1、PWM 通道与引脚

RK3568 有 4 个 PWM 模块，每个 PWM 模块有 4 个通道，因此一共有 16 路 PWM：PWM0~PWM15，PWM 模块与 PWM 通道以及对应的引脚关系如表 23.1.1.1 所示：

PWM 模块	通道	标号	可用的引脚
PWM0	CH0	PWM0	GPIO0_B7/PWM0_M0
			GPIO0_C7/PWM0_M1
	CH1	PWM1	GPIO0_C0/PWM1_M0
			GPIO0_B5/PWM1_M1
CH2	PWM2	GPIO0_C1/PWM2_M0	
		GPIO0_B6/PWM2_M1	
CH3	PWM3	GPIO0_C2/PWM3_IR	
PWM1	CH0	PWM4	GPIO0_C3/PWM4
	CH1	PWM5	GPIO0_C4/PWM5
	CH2	PWM6	GPIO0_C5/PWM6
	CH3	PWM7	GPIO0_C6/PWM7_IR

PWM2	CH0	PWM8	GPIO3_B1/PWM8_M0
			GPIO1_D5/PWM8_M1
	CH1	PWM9	GPIO3_B2/PWM9_M0
			GPIO1_D6/PWM9_M1
	CH2	PWM15	GPIO3_B5/PWM15_M0
			GPIO2_A1/PWM15_M1
	CH3	PWM11	GPIO3_B6/PWM11_IR_M0
			GPIO4_C0/PWM11_IR_M1
PWM3	CH0	PWM12	GPIO3_B7/ PWM12_M0
			GPIO4_C5/ PWM12_M1
	CH1	PWM13	GPIO3_C0/ PWM13_M0
			GPIO4_C6/ PWM13_M1
	CH2	PWM14	GPIO3_C4/ PWM14_M0
			GPIO4_C2/ PWM14_M1
	CH3	PWM15	GPIO3_C5/PWM15_IR_M0
			GPIO4_C3/PWM15_IR_M1

图 23.1.1.1 RK3568 PWM 通道与引脚

其中 PWM3, PWM7, PWM11 和 PWM15 可用于 IR。

2、PWM 简介

RK3568 有 4 个 PWM 模块，每个 PWM 模块的功能基本相同，这些 PWM 通道的特性如下：

- ①、每个 PWM 模块有 4 个通道。
- ②、支持捕获模式：
 - 测量输入波形高低电平的有效周期。
 - 输入波形极性变化的时候产生中断信号。
 - 32 位高电平捕获寄存器。
 - 32 位低电平捕获寄存器。
 - 32 位当前值寄存器。
 - 捕获结果可以保存到 FIFO 中，FIFO 深度为 8，FIFO 中的数据可以通过 CPU 或 DMA 读取。
- ③、支持连续以及单次模式：
 - 32 位的周期计数寄存器。
 - 32 位的占空比寄存器。
 - 32 位当前值寄存器。
 - 输出 PWM 极性以及占空比可调。
 - 可配置中央对齐或左对齐模式。

3、PWM 设备节点

接下来看一下 PWM 的设备树，RK3568 的 PWM 设备树绑定信息文档为：[Documentation/devicetree/bindings/pwm/pwm-rockchip.txt](#)，我们简单总结一下 PWM 节点信息。

①、必须的参数：

compatible: 必须是“rockchip,<name>-pwm”形式的，比如：rockchip,rk3288-pwm、rock

chip,rk3568-pwm 等。对于 RK3568 而言，有效的是 rockchip,rk3288-pwm。

reg: PWM 控制器物理寄存器基地址，比如对于 PWM0 来说，这个地址为 0XFDD70000，这个可以在 RK3568 的数据手册上找到。

clocks: 时钟源。

#pwm-cells: 瑞芯微的芯片必须是 2 或者 3，对于 RK3568 来说是 3。

了解完 PWM 的绑定文档以后，我们来看一下 RK3568 实际的定时器节点，打开 rk3568.dtsi，找到名为“pwm15”的设备节点，内容如下：

示例代码 23.1.1.1 PWM15 节点

```

1 pwm15: pwm@fe700030 {
2     compatible = "rockchip,rk3568-pwm", "rockchip,rk3328-pwm";
3     reg = <0x0 0xfe700030 0x0 0x10>;
4     interrupts = <GIC_SPI 85 IRQ_TYPE_LEVEL_HIGH>,
5                 <GIC_SPI 89 IRQ_TYPE_LEVEL_HIGH>;
6     #pwm-cells = <3>;
7     pinctrl-names = "active";
8     pinctrl-0 = <&pwm15m0_pins>;
9     clocks = <&cru CLK_PWM3>, <&cru PCLK_PWM3>;
10    clock-names = "pwm", "pclk";
11    status = "disabled";
12};
    
```

RK3568 的 PWM 节点的 compatible 属性为“rockchip,rk3568-pwm”和“rockchip,rk3328-pwm”，我们可以在 linux 内核源码中搜索这两个字符串就可以找到 PWM 驱动文件，这个文件为：drivers/pwm/pwm-rockchip.c。

23.1.2 PWM 子系统

Linux 内核提供了个 PWM 子系统框架，编写 PWM 驱动的时候一定要符合这个框架。PWM 子系统的核心是 pwm_chip 结构体，定义在文件 include/linux/pwm.h 中，定义如下：

示例代码 23.1.2.1 pwm_chip 结构体

```

1 struct pwm_chip {
2     struct device      *dev;
3     struct list_head  list;
4     const struct pwm_ops *ops;
5     int                base;
6     unsigned int      npwm;
7
8     struct pwm_device *pwms;
9
10    struct pwm_device * (*of_xlate)(struct pwm_chip *pc,
11                                   const struct of_phandle_args *args);
12    unsigned int      of_pwm_n_cells;
13    bool              can_sleep;
14 };
    
```

第 4 行，pwm_ops 结构体就是 PWM 外设的各种操作函数集合，编写 PWM 外设驱动的时

候需要开发人员实现。pwm_ops 结构体也定义在 pwm.h 头文件中，定义如下：

示例代码 23.1.2.2 pwm_ops 结构体

```

1 struct pwm_ops {
2     int (*request)(struct pwm_chip *chip, //请求 PWM
3                 struct pwm_device *pwm);
4     void (*free)(struct pwm_chip *chip, //释放 PWM
5                struct pwm_device *pwm);
6     int (*config)(struct pwm_chip *chip, //配置 PWM 周期和占空比
7                  struct pwm_device *pwm,
8                  int duty_ns, int period_ns);
9     int (*set_polarity)(struct pwm_chip *chip, //设置 PWM 极性
10                        struct pwm_device *pwm,
11                        enum pwm_polarity polarity);
12    int (*enable)(struct pwm_chip *chip, //使能 PWM
13                struct pwm_device *pwm);
14    void (*disable)(struct pwm_chip *chip, //关闭 PWM
15                  struct pwm_device *pwm);
16 #ifdef CONFIG_DEBUG_FS
17    void (*dbg_show)(struct pwm_chip *chip,
18                    struct seq_file *s);
19 #endif
20    struct module *owner;
21 };
    
```

pwm_ops 中的这些函数不一定全部实现，但是像 config、enable 和 disable 这些肯定是需要实现的，否则的话打开/关闭 PWM，设置 PWM 的占空比这些就没操作了。

PWM 子系统驱动的核心初始化 pwm_chip 结构体，然后向内核注册初始化完成以后的 pwm_chip。这里就要用到 pwmchip_add 函数，此函数定义在 drivers/pwm/core.c 文件中，函数原型如下：

```
int pwmchip_add(struct pwm_chip *chip)
```

函数参数和返回值含义如下：

chip: 要向内核注册的 pwm_chip。

返回值: 0 成功；负数 失败。

卸载 PWM 驱动的时候需要将前面注册的 pwm_chip 从内核移除掉，这里要用到 pwmchip_remove 函数，函数原型如下：

```
int pwmchip_remove(struct pwm_chip *chip)
```

函数参数和返回值含义如下：

chip: 要移除的 pwm_chip。

返回值: 0 成功；负数 失败。

23.1.3 PWM 驱动源码分析

我们简单分析一下 Linux 内核自带的 RK3568 PWM 驱动，驱动文件前面都说了，是 pwm-rockchip.c 这个文件。打开这个文件，可以看到，这是一个标准的平台设备驱动文件，有如下所示：

示例代码 23.1.3.1 RK3568 PWM 平台驱动

```

1 static const struct of_device_id rockchip_pwm_dt_ids[] = {
2     { .compatible = "rockchip,rk2928-pwm", .data = &pwm_data_v1},
3     { .compatible = "rockchip,rk3288-pwm", .data = &pwm_data_v2},
4     { .compatible = "rockchip,vop-pwm", .data = &pwm_data_vop},
5     { /* sentinel */ }
6 };
7 .....
8 static struct platform_driver rockchip_pwm_driver = {
9     .driver = {
10        .name = "rockchip-pwm",
11        .of_match_table = rockchip_pwm_dt_ids,
12    },
13    .probe = rockchip_pwm_probe,
14    .remove = rockchip_pwm_remove,
15 };
16 module_platform_driver(rockchip_pwm_driver);
    
```

第 2~4 行, 当设备树 PWM 节点的 compatible 属性值为“rockchip,rk2928-pwm”、“rockchip,rk3288-pwm”和“rockchip,vop-pwm”中的某一个的话就会匹配此驱动。

第 13 行, 当设备树节点和驱动匹配以后 rockchip_pwm_probe 函数就会执行。

在看 rockchip_pwm_probe 函数之前先来看下 rockchip_pwm_chip stm32_pwm 结构体, 这个结构体是瑞芯微官方创建的针对瑞芯微芯片的 PWM 结构体, 这个结构体会贯穿整个 PWM 驱动, 起到灵魂的作用。rockchip_pwm_chip 结构体内容如下:

示例代码 23.1.3.2 rockchip_pwm_chip 结构体

```

1 struct rockchip_pwm_chip {
2     struct pwm_chip chip;
3     struct clk *clk;
4     const struct rockchip_pwm_data *data;
5     void __iomem *base;
6 };
    
```

重点看一下第 2 行, 这是一个 pwm_chip 结构体成员变量 chip, 前面说了, PWM 子系统的核心就是 pwm_chip。

rockchip_pwm_probe 函数如下(有缩减):

示例代码 23.1.3.3 rockchip_pwm_probe 函数

```

1 static int rockchip_pwm_probe(struct platform_device *pdev)
2 {
3     const struct of_device_id *id;
4     struct rockchip_pwm_chip *pc;
5     struct resource *r;
6     int ret;
7
8     id = of_match_device(rockchip_pwm_dt_ids, &pdev->dev);
9     if (!id)
    
```

```

10     return -EINVAL;
11
12     pc = devm_kzalloc(&pdev->dev, sizeof(*pc), GFP_KERNEL);
13     if (!pc)
14         return -ENOMEM;
15
16     r = platform_get_resource(pdev, IORESOURCE_MEM, 0);
17     pc->base = devm_ioremap_resource(&pdev->dev, r);
18     if (IS_ERR(pc->base))
19         return PTR_ERR(pc->base);
20
21     pc->clk = devm_clk_get(&pdev->dev, NULL);
22     if (IS_ERR(pc->clk))
23         return PTR_ERR(pc->clk);
24
25     ret = clk_prepare(pc->clk);
26     if (ret)
27         return ret;
28
29     platform_set_drvdata(pdev, pc);
30
31     pc->data = id->data;
32     pc->chip.dev = &pdev->dev;
33     pc->chip.ops = pc->data->ops;
34     pc->chip.base = -1;
35     pc->chip.npwm = 1;
36
37     if (pc->data->ops->set_polarity) {
38         pc->chip.of_xlate = of_pwm_xlate_with_flags;
39         pc->chip.of_pwm_n_cells = 3;
40     }
41
42     ret = pwmchip_add(&pc->chip);
43     if (ret < 0) {
44         clk_unprepare(pc->clk);
45         dev_err(&pdev->dev, "pwmchip_add() failed: %d\n", ret);
46     }
47
48     return ret;
49 }
    
```

第 12 行, `pc` 是一个 `rockchip_pwm_chip` 类型的结构体指针变量, 这里为其申请内存。`rockchip_pwm_chip` 结构体有个重要的成员变量 `chip`, `chip` 是 `pwm_chip` 类型的。所以这一行就引出了 PWM 子系统核心部件 `pwm_chip`, 稍后的重点就是初始化 `chip`。

第 32~35 行, 初始化 pc 的 chip 成员变量, 也就是初始化 pwm_chip! 第 33 行设置 pwm_chip 的 ops 操作集为 pc->data->ops。根据示例代码 23.1.3.1 中的 compatible 属性可以, RK3568 对应的 data 是 pwm_data_v2, 因此 pwm_data_v2 源码如下:

示例代码 23.1.3.4 pwm_data_v2 源码

```

1 static const struct rockchip_pwm_data pwm_data_v2 = {
2     .regs = {
3         .duty = 0x08,
4         .period = 0x04,
5         .cntr = 0x00,
6         .ctrl = 0x0c,
7     },
8     .prescaler = 1,
9     .ops = &rockchip_pwm_ops_v2,
10    .set_enable = rockchip_pwm_set_enable_v2,
11 };
    
```

第 9 行, rockchip_pwm_ops_v2 就是 pwm_chip 的 ops 函数, 也就是 RK3568 的 ops 函数。

继续回到示例代码 23.1.3.3 中, 第 49 行使用 pwmchip_add 函数向内核添加 pwm_chip, 这个就是 rockchip_pwm_probe 函数的主要工作。

我们重点来看一下 rockchip_pwm_ops_v2, 定义如下:

示例代码 23.1.3.5 rockchip_pwm_ops_v2 操作集合

```

1 static const struct pwm_ops rockchip_pwm_ops_v2 = {
2     .config = rockchip_pwm_config,
3     .set_polarity = rockchip_pwm_set_polarity,
4     .enable = rockchip_pwm_enable,
5     .disable = rockchip_pwm_disable,
6     .owner = THIS_MODULE,
7 };
    
```

第 2 行 rockchip_pwm_config 就是最终的 PWM 设置函数, 我们在应用中设置的 PWM 频率和占空比最终就是由 rockchip_pwm_config 函数来完成的, 此函数会最终操作 RK3568 相关的寄存器。

rockchip_pwm_config 函数源码如下:

示例代码 23.1.3.6 rockchip_pwm_config 函数

```

1 static int rockchip_pwm_config(struct pwm_chip *chip,
2     struct pwm_device *pwm, int duty_ns, int period_ns)
3 {
4     struct rockchip_pwm_chip *pc = to_rockchip_pwm_chip(chip);
5     unsigned long period, duty;
6     u64 clk_rate, div;
7     int ret;
8
9     clk_rate = clk_get_rate(pc->clk);
10
11     /*
    
```

```

12  * Since period and duty cycle registers have a width of 32
13  * bits, every possible input period can be obtained using the
14  * default prescaler value for all practical clock rate values.
15  */
16  div = clk_rate * period_ns;
17  do_div(div, pc->data->prescaler * NSEC_PER_SEC);
18  period = div;
19
20  div = clk_rate * duty_ns;
21  do_div(div, pc->data->prescaler * NSEC_PER_SEC);
22  duty = div;
23
24  ret = clk_enable(pc->clk);
25  if (ret)
26      return ret;
27
28  writel(period, pc->base + pc->data->regs.period);
29  writel(duty, pc->base + pc->data->regs.duty);
30  writel(0, pc->base + pc->data->regs.cntr);
31
32  clk_disable(pc->clk);
33
34  return 0;
35 }
    
```

第 16~18 行，根据设置的频率，计算出 RK3568 PWM 的 PERIOD 寄存器。

第 20~22 行，根据设置的占空比，计算出 DUTY 寄存器的值。

第 28~30 行，设置 PWM 外设的 PERIOD、DUTY 和 CNT 寄存器。

至此，RK3568 的 PWM 驱动基本分析到这里。

23.2 PWM 驱动编写

23.2.1 修改设备树

PWM 驱动就不需要我们再编写了，瑞芯微已经写好了，前面我们也已经详细的分析过这个驱动源码了。我们在实际使用的时候只需要修改设备树即可，ATK-DLRK3568 开发板上的 JP11 排针引出了 GPIO3_C5 这个引脚，这个引脚可以用作 PWM15_IR_M0，如图 23.2.1.1 所示：

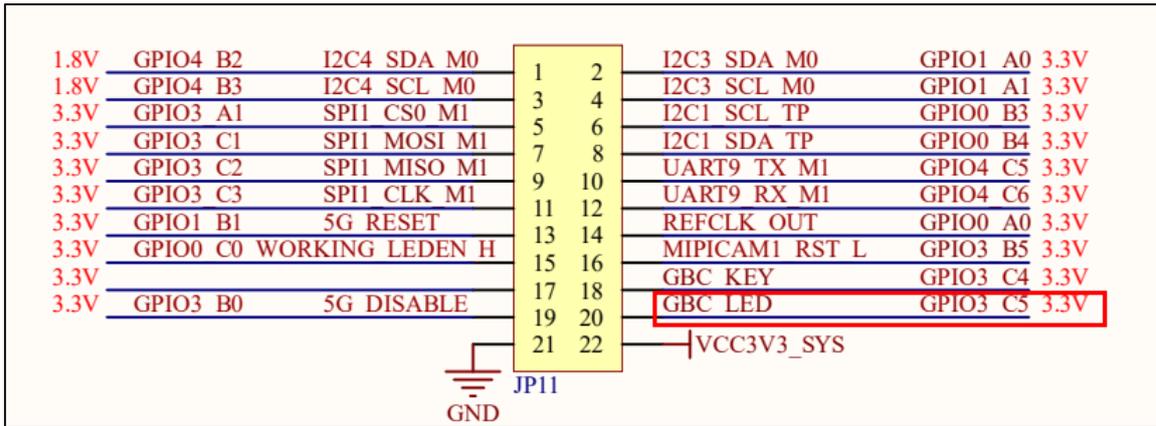


图 23.2.1.1 GPIO3_C5 引脚

GPIO3_C5 可以作为 PWM3 的通道 3 的 PWM 输出引脚，也就是 PWM15_IR_M0，所以我们需要在设备树里面添加 GPIO3_C5 的引脚信息以及 PWM15 外设信息。

1、添加 GPIO3_C5 引脚信息

PWM 的引脚配置瑞芯微已经帮我们做好了，打开 rk3568-pinctrl.dtsi 文件，找到如下所示内容：

```

示例代码 23.2.1.1 TIM1 PWM 引脚信息
1 pwm15 {
2     /omit-if-no-ref/
3     pwm15m0_pins: pwm15m0-pins {
4         rockchip,pins =
5             /* pwm15_irm0 */
6             <3 RK_PC5 1 &pcfg_pull_down>;
7     };
8
9     /omit-if-no-ref/
10    pwm15m1_pins: pwm15m1-pins {
11        rockchip,pins =
12            /* pwm15_irm1 */
13            <4 RK_PC3 1 &pcfg_pull_none>;
14    };
15};

```

可以看出瑞芯微官方对 GPIO3_C5 写了两个配置，第 6 行是把 GPIO3_C5 配置为 PWM15_IR_M0 引脚，但是默认没有上下拉！大家根据实际情况修改为上下拉配置即可。本章实验，我们选择 pcfg_pull_down，也就是将 GPIO3_C5 配置为 PWM15_IR_M0，默认下拉。

2、向 pwm15 节点追加信息

前面已经讲过了，rk3568.dtsi 文件中已经有了“pwm15”节点，但是这个节点默认是 disable 的，还不能直接使用，所以需要在 rk3568-atk-evb1-ddr4-v10.dtsi 文件中向打开 pwm15 节点。

```

示例代码 23.2.1.3 向 timers1 添加的内容
1 &pwm15 {
2     status = "okay";

```

3}

第 2 行, status 改为 okay, 也就是使能 PWM15。

23.2.2 使能 PWM 驱动

瑞芯微官方的 Linux 内核已经默认使能了 PWM 驱动, 所以不需要我们修改, 但是为了学习, 我们还是需要知道怎么使能。打开 Linux 内核配置界面, 按照如下路径找到配置项:

```
-> Device Drivers |
    -> Pulse-Width Modulation (PWM) Support (PWM [=y])
        -> <*> Rockchip PWM support //选中
```

配置如图 23.2.2.1 所示:

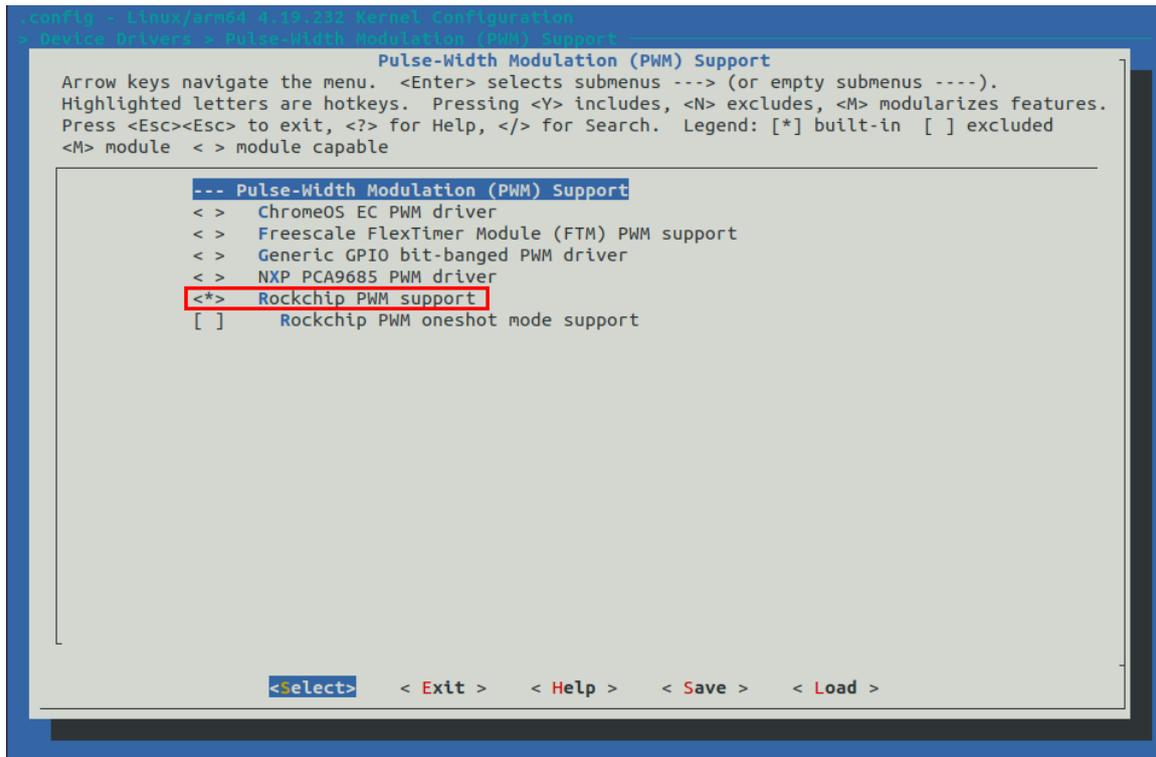


图 23.2.2.1 PWM 配置项

23.3 PWM 驱动测试

1、确定 PWM15 对应的 pwmchipX 文件

使用新的设备树启动系统, 然后将开发板上的 GPIO3_C5 引脚连接到示波器上, 通过示波器来查看 PWM 波形图。我们可以直接在用户层来配置 PWM, 进入目录/sys/class/pwm 中, 如图 23.3.1 所示:

```
root@ATK-DLRK356X:/# cd /sys/class/pwm
root@ATK-DLRK356X:/sys/class/pwm# ls
pwmchip0 pwmchip1 pwmchip2 pwmchip3
root@ATK-DLRK356X:/sys/class/pwm#
```

图 23.3.1 当前系统下的 PWM 外设

注意! 图 23.3.1 中有个 pwmchip0~pwmchip3, 但是我们并不知道哪个是 PWM15 的。我们可以通过查看 pwmchip0~pwmchip3 的外设基地址哪个和 PWM15 一样, 那么哪个就是 PWM15

的。

ls -l

进入到 pwm 执行上面的 ls -l 指令，以后会打印出其链接的路径，如图 23.3.2 所示：

```
root@ATK-DLRK356X:/sys/class/pwm# ls -l
total 0
lrwxrwxrwx 1 root root 0 Jun  7 09:55 pwmchip0 -> ../../devices/platform/fdd70020.pwm/pwm/pwmchip0
lrwxrwxrwx 1 root root 0 Jun  7 09:55 pwmchip1 -> ../../devices/platform/fe6e0000.pwm/pwm/pwmchip1
lrwxrwxrwx 1 root root 0 Jun  7 09:55 pwmchip2 -> ../../devices/platform/fe6e0010.pwm/pwm/pwmchip2
lrwxrwxrwx 1 root root 0 Jun  7 09:55 pwmchip3 -> ../../devices/platform/fe700030.pwm/pwm/pwmchip3
root@ATK-DLRK356X:/sys/class/pwm#
```

PWM15 外设基地址

图 23.3.2 pwmchip1 路径名称

从图 23.3.2 可以看出 pwmchip3 对应的定时器寄存器起始地址为 0XFE700030，根据示例代码 23.1.1.1 中的 pwm15 节点，可以知道 PWM15 这个定时器的寄存器起始地址就是 0XFE700030。因此，pwmchip1 就是 PWM15 对应的文件。

为什么要用这么复杂的方式来确定定时器对应的 pwmchip 文件呢？因为当 RK3568 开启多个 PWM 以后，其 pwmchip 文件就会变！

2、调出 pwmchip15 的 pwm0 子目录

输入如下命令打开 pwmchip15 的 pwm0 子目录

```
echo 0 > /sys/class/pwm/pwmchip3/export
```

执行完成会在 pwmchip3 目录下生成一个名为“pwm0”的子目录，如图 23.3.3 所示：

```
root@ATK-DLRK356X:/sys/class/pwm/pwmchip3# ls
device export npwm power pwm0 subsystem uevent unexport
root@ATK-DLRK356X:/sys/class/pwm/pwmchip3#
```

图 23.3.3 新生成的 pwm0 子目录

2、设置 PWM 的频率

注意，这里设置的是周期值，单位为 ns，比如 20KHz 频率的周期就是 50000ns，输入如下命令：

```
echo 50000 > /sys/class/pwm/pwmchip3/pwm0/period
```

3、设置 PWM 的占空比

这里不能直接设置占空比，而是设置的一个周期的 ON 时间，也就是高电平时间，比如 20KHz 频率下 20%占空比的 ON 时间就是 10000，输入如下命令：

```
echo 10000 > /sys/class/pwm/pwmchip3/pwm0/duty_cycle
```

4、设置 PWM 极性

设置一下 PWM 波形的极性，输入如下命令：

```
echo normal > /sys/class/pwm/pwmchip3/pwm0/polarity
```

极性设置为 normal，也就是 duty_cycle 为高电平时间。如果要将极性反过来，可以设置为 inversed。

5、使能 PWM

一定要先设置频率和波特率，最后在开启 PWM，否则会提示参数错误！输入如下命令使能 PWM：

```
echo 1 > /sys/class/pwm/pwmchip3/pwm0/enable
```

设置完成使用示波器查看波形是否正确，正确的话如图 23.3.4 所示：

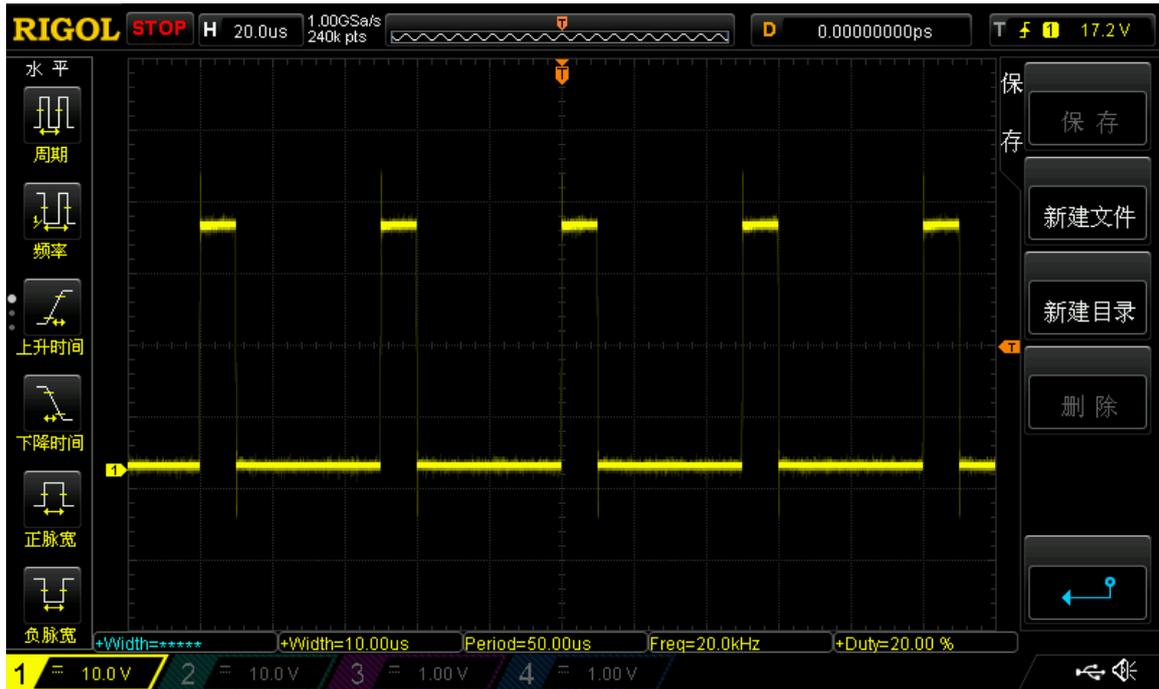


图 23.3.4 PWM 波形图

从图 23.3.4 可以看出，此时 PWM 频率为 20KHz，占空比为 20%，与我们设置的一致。如果要修改频率或者占空比的话一定要注意这两者时间值，比如 20KHz 频率的周期值为 50000ns，那么你在调整占空比的时候 ON 时间就不能设置大于 50000，否则就会提示你参数无效。

第二十四章 MIPI DSI 屏幕驱动实验

MIPI DSI 屏幕目前广泛应用于手机、平板等产品中，尤其是高清屏幕基本都是采用 MIPI DSI 接口，比如 1080P、2K 级的屏幕。MIPI DSI 接口使用更少的线数，驱动更高分辨率的屏幕。一般低端 ARM 芯片，会提供 RGB 接口来驱动 LCD，中高端 ARM 芯片会提供 MIPI DSI 接口，比如手机、平板的 SOC。ATK-DLRK3568 也提供了一路 MIPI DSI 接口，本章我们就来学习如何调试 MIPI DSI 接口屏幕。

本章节用到的所有 MIPI 协议文档已经放到了开发板光盘中，路径为：**开发板光盘**→**06、参考资料**→**MIPI 协议文档**。

24.1 MIPI 联盟简介

MIPI 即移动产业处理器接口，是 Mobile Industry Processor Interface 缩写，2003 年由 ARM、Nokia、ST 和 TI 等公司成立的一个联盟，所以大家在看 MIPI 相关文档的时候长看到 MIPI Alliance。MIPI 官方网址为 <https://www.mipi.org/>。当前 MIPI 联盟已经有很多成员，比如大家熟知的 ARM、ST、高通、TI、苹果、海思等，基本上囊括了主流的移动处理器生产厂家，如图 24.1.1 所示：



图 24.1.1 MIPI 联盟成员(部分)

MIPI 联盟主要是为移动处理器定制标准接口和规范，开发的接口广泛应用于处理器、相机、显示屏、基带调制解调器等设备。比如：

- MIPI DSI(显示屏接口)
- MIPI CSI(摄像头接口)
- MIPI I3C
- MIPI RFFE(射频前端控制接口)
- MIPI SPMI(系统电源管理接口)

• 等等。

MIPI 通过定义一套协议和标准，满足各个子系统之间互联，确保不同公司开发的产品可以兼容连接，减少协议标准。图 24.1.2 就是 MIPI 相关协议在移动处理器中的使用：

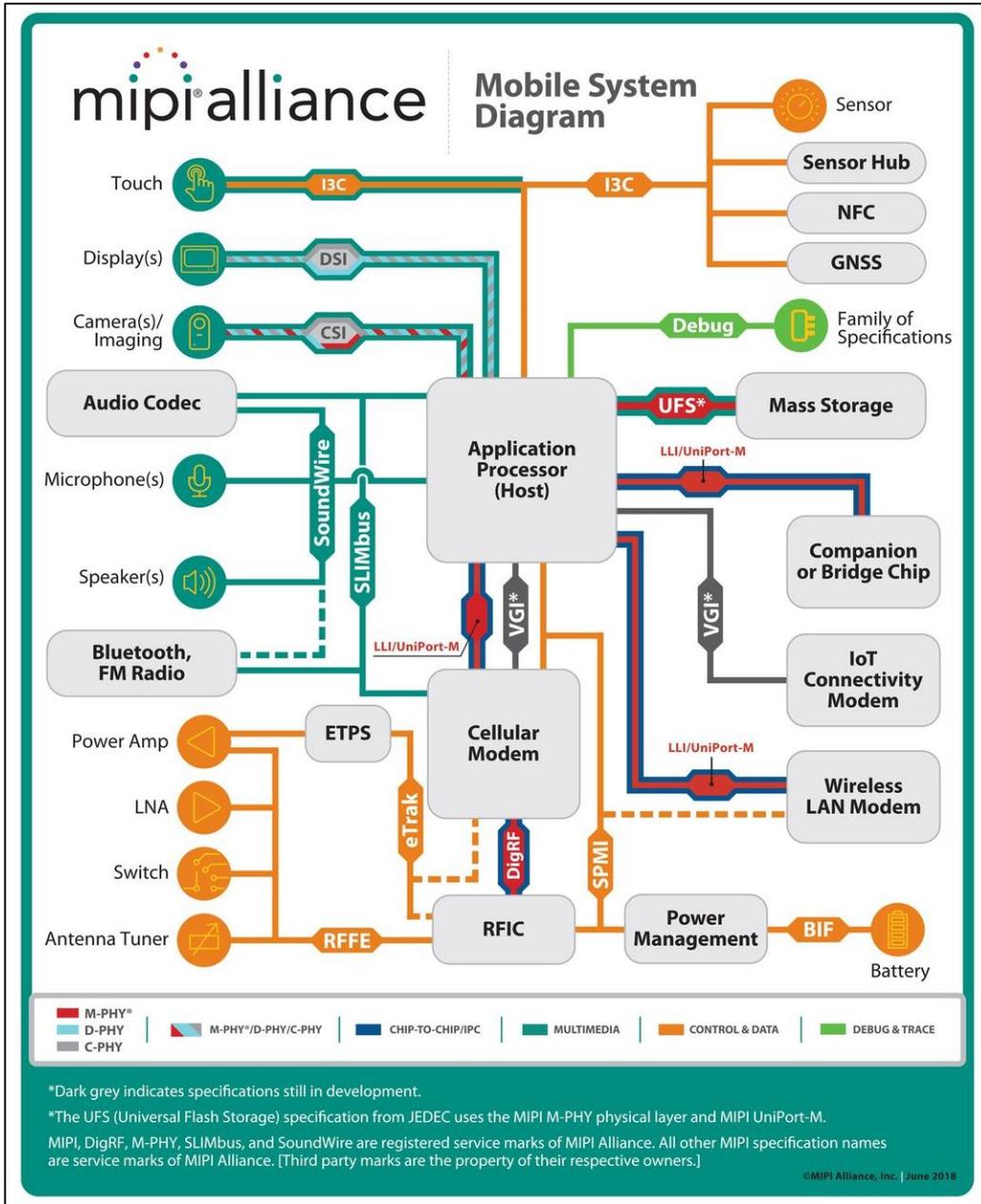


图 24.1.2 移动处理器中 MIPI 的使用场合

从图 24.1.2 可以看出，MIPI 下的各种协议、接口在移动处理器中使用非常广泛，典型的的就是 MIPI DSI 和 MIPI CSI。MIPI 主要包括四个方面，如图 24.1.3 所示：

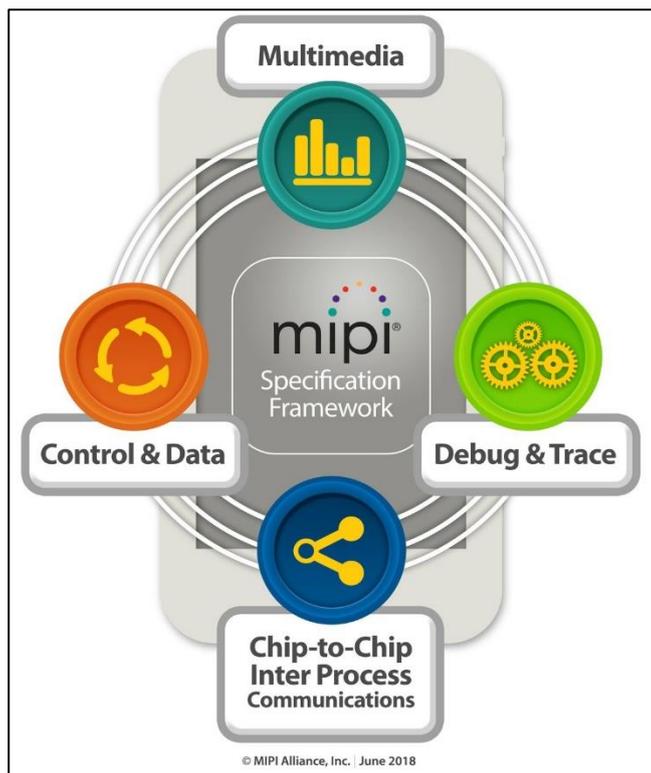


图 24.1.3 MIPI 框架

从图 24.1.3 可以看出，MIPI 主要有四个方向的协议：

- ①、Multimedia，多媒体。
- ②、Control&Data，控制和数据。
- ③、Chip-to-Chip Inter Process Communications，
- ④、Debug&Trace，调试和追踪

我们依次来简单看一下这四部分都有哪些内容。

1、Multimedia

Multimedia 部分框图如图 24.1.4 所示：

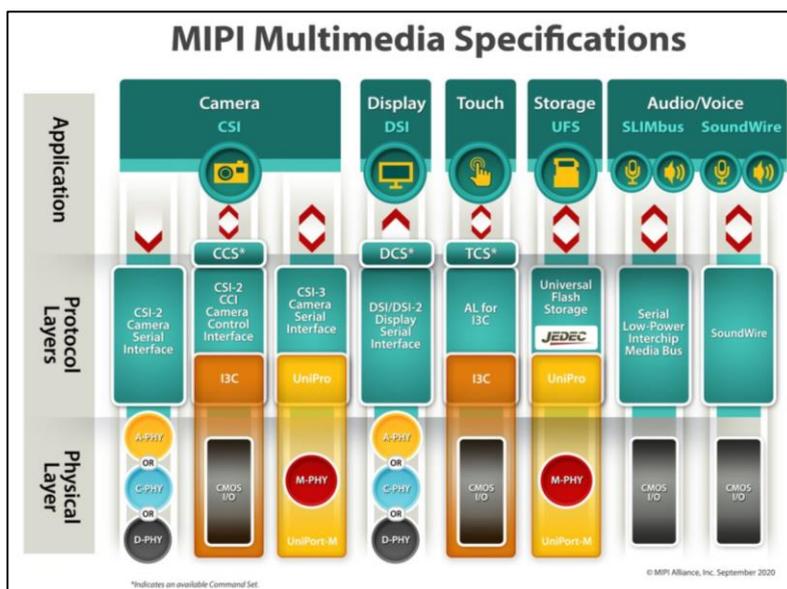


图 24.1.4 Multimedia 部分框图

Multimedia 就是多媒体部分，分为如下几部分：

- 摄像头，应用层有 CCS，协议层主要有 CSI-2、CSI-3，物理层有 A-PHY、C-PHY、D-PHY 和 M-PHY。
- 屏幕，应用层有 DCS，协议层主要有 DSI，物理层有 A-PHY、C-PHY、D-PHY。
- 触摸，应用层有 TCS，协议层是 I3C。
- 存储，UFS 协议，这个是目前手机以及平板上最常用的存储协议，物理层为 M-PHY。
- 音频，协议层有 SLIMbus 和 SoundWire。

多媒体部分我们用的最多就是 DSI 和 CSI，DSI 应用于屏幕，CSI 用于摄像头。对应的物理层协议有 A-PHY、C-PHY、D-PHY 和 M-PHY。

D-PHY：目前用的最多的接口，不管是摄像头还是屏幕，D-PHY 接口为 1/2/4lane(lane 可以理解为通道，也就是 1/2/3/4 通道，每个通道 2 条差分线)，外加一对时钟线，数据线和时钟线都是差分线，为电流驱动型，不同版本的 D-PHY 速度不同，比如 RK3568 用的 V1.2 版本的 D-PHY 单 lane 最高可到 2.5Gbps(使用 deskew 校准)。D-PHY 最多 10 根线，有专门的时钟线来进行同步。

C-PHY：随着屏幕和摄像头的分辨率以及帧率越来越高，D-PHY 的带宽越来越不够用。C-PHY 应运而生，C-PHY 接口是 1/2/3 Trio，每个 Trio 有 3 根线，最高 9 根线，没有专用的时钟线了。C-PHY 目前在高端旗舰手机芯片中可能会用到，本教程不讲解 C-PHY。

A-PHY：主要为汽车自动驾驶而生，目前汽车自动驾驶发展非常迅猛，ADAS（高级驾驶员辅助系统）摄像头于车载娱乐屏幕越来越多，分辨率也越来越高，而且车载摄像头和娱乐屏幕分布比较分散，到主控的距离一般比较长。但是 C-PHY 和 D-PHY 的距离太短，最多不超过 15CM，显然不适合用在当今高度智能化的车载领域。A-PHY 于 2020 年 9 月发布，用于长距离、超高速的汽车应用中，比如 ADAS、自动驾驶系统（ADS）、车载信息娱乐系统（IVI）和其他环绕传感器。

M-PHY：目前主要用在 USF 存储中。

2、Control&Data

Control&Data 部分框图如图 24.1.5 所示：

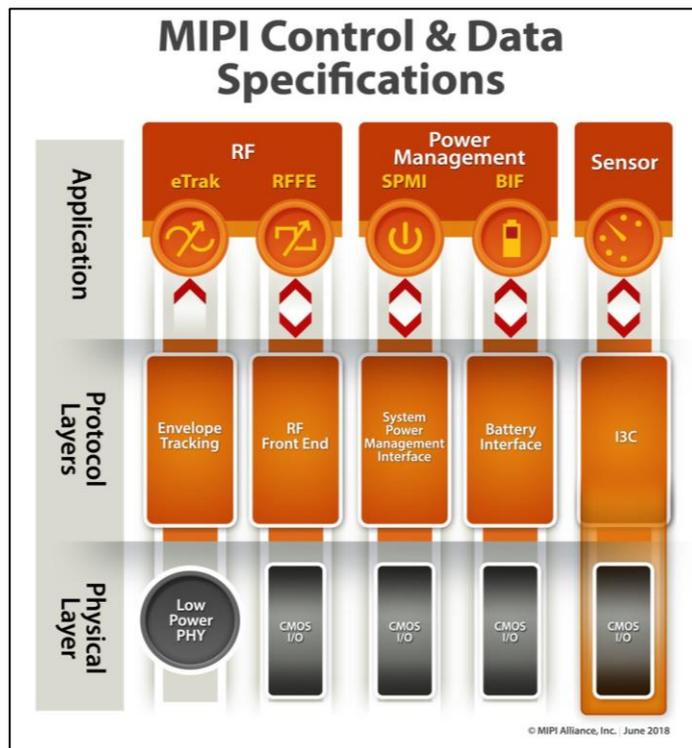


图 24.1.5 Control&Data

图 24.1.5 中主要是 RF、电源管理以及 I3C 通信接口相关的协议。

3、Chip-to-Chip Inter Process Communications

Chip-to-Chip Inter Process Communications 框图如图 24.1.6 所示：

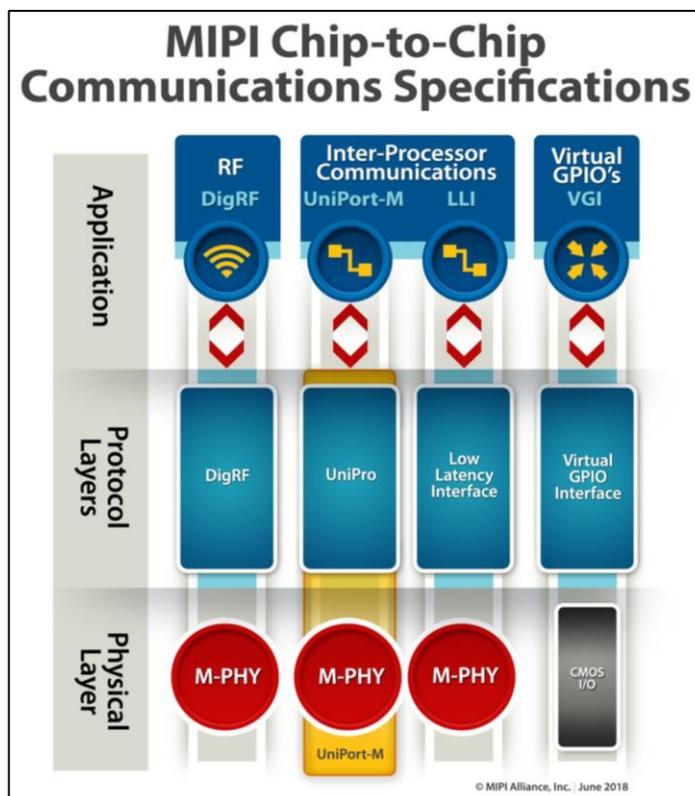


图 24.1.6 Chip-to-Chip Inter Process Communications 框图

4、Debug&Trace

Debug&Trace 框图如图 24.1.7 所示:

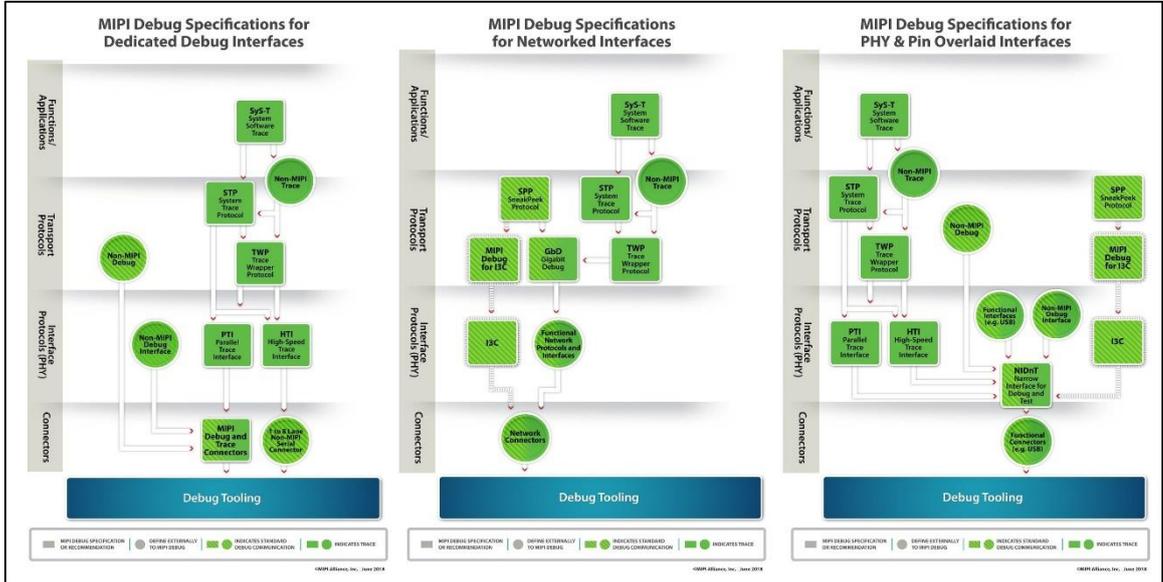


图 24.1.7 Debug&Trace 框图

关于 MIPI 联盟就介绍到这里，感兴趣的可以去 MIPI 联盟官网上去看一下。

24.2 MIPI DSI 概述

24.2.1 MIPI DSI 协议综述

本章我们是来学习如何驱动 MIPI 接口屏幕，所以需要学习的就是 MIPI DSI。DSI 全称是 Display Serial Interface，是主控和显示模组之间的串行连接接口，图 24.2.1.1 展示了主控和屏幕之间的连接方式：

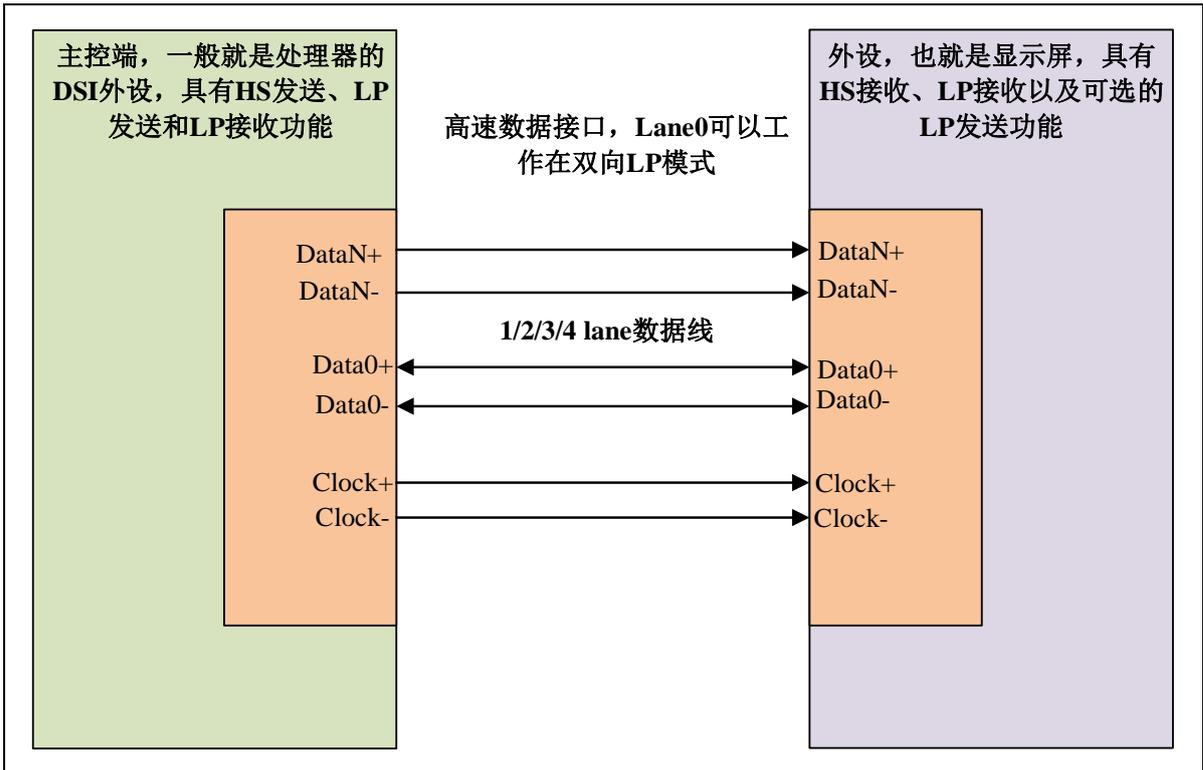


图 24.2.1.1 DSI 主控与屏幕之间接口示意图

MIPI DSI 接口分为数据线和时钟线，均为差分信号。数据线可选择 1/2/3/4 lanes，时钟线有一对，最多 10 根线。MIPI DSI 以串行的方式发送指令和数据给屏幕，也可以读取屏幕中的信息。很明显，如果屏幕的分辨率和帧率越高，需要的带宽就越大，就需要更多的数据线来传输图像数据，RK3568 默认使用 4 lanes 来驱动 MIPI 屏幕，当然了，对于小尺寸的屏幕，也可以使用 2 lanes 来驱动。对于 MIPI DSI 接口而言，最常用的就是 2 lanes 和 4 lanes。

24.2.2 MIPI DSI 分层

和网络协议栈一样，MIPI DSI 也是分层的，如图 24.2.2.1 所示：

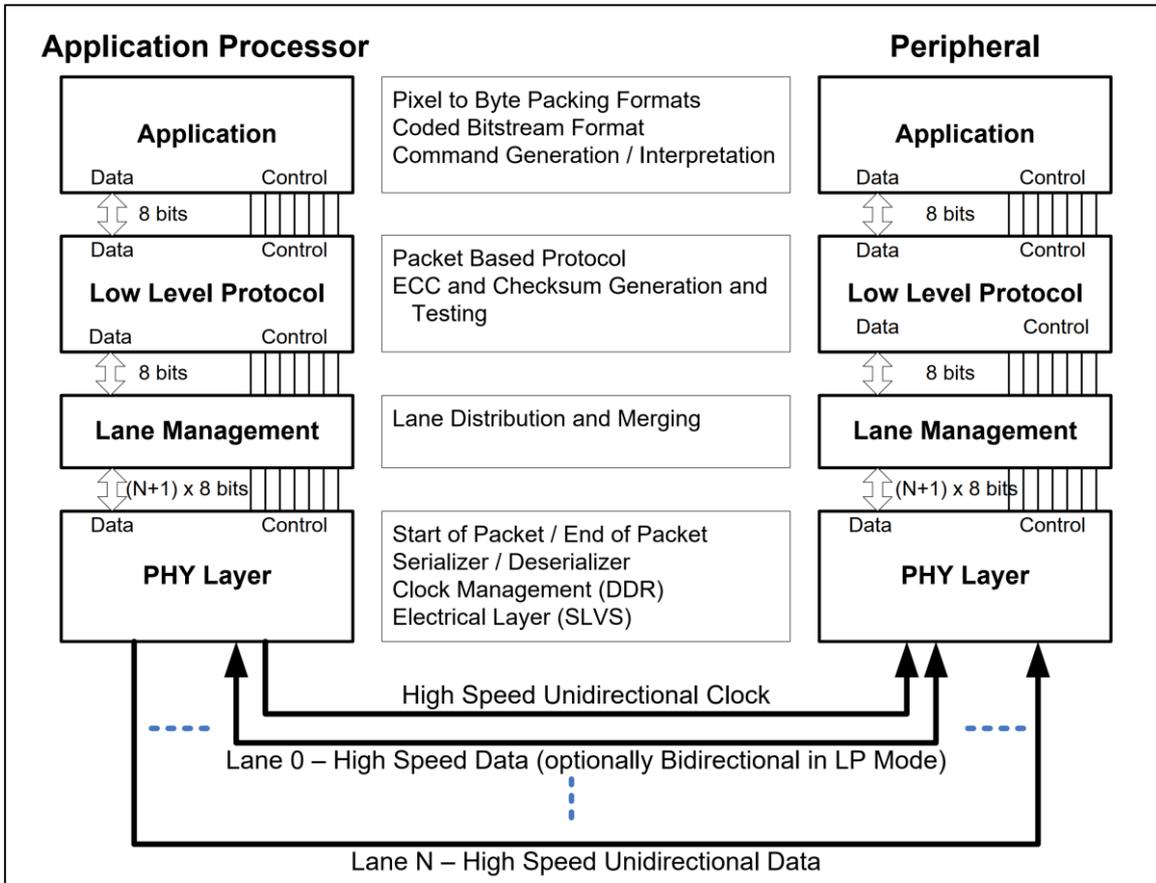


图 24.2.2.1 MIPI DSI 协议分层

从图 24.2.2.1 可以看出，MIPI DSI 一共有四层，从上往下依次为：

- 应用层
- 协议层
- 通道管理层
- 物理层

1、应用层

应用层处理更高层次的编码，将要显示的数据打包进数据流中，下层会处理并发送应用层的数据流。发送端将命令以及数据编码成 MIPI DSI 规格的格式，接收端则将接收到的数据还原为原始的数据。

2、协议层

协议层主要是打包数据，在原始的数据上添加 ECC 和校验和等东西。应用层传递下来的数据会打包成两种格式的数据：长数据包和短数据包，关于长短数据包后面会有详细讲解。发送端将原始数据打包好，添加包头和包尾，然后将打包好的数据发送给下层。接收端收到下层传来的数据包以后执行相反的操作，去除包头和包尾，然后使用 ECC 进行校验接收到的数据，如果没问题就将解包后的原始数据交给应用层。

3、链路层

链路层负责如何将数据分配到具体的通道上，MIPI DSI 可以支持 1/2/3/4 Lane，采用几通道取决于你的实际应用，如果带宽需求低，那么 2 Lane 就够了，带宽高的话就要 4 Lane。协议层

下来的数据包都是串行的, 如果只有 1 Lane 的话, 那就直接使用这 1 Lane 将数据串行的发送出去, 如果是 2/4 Lane 的话数据该如何发送呢? 如图 24.2.2.2 所示:

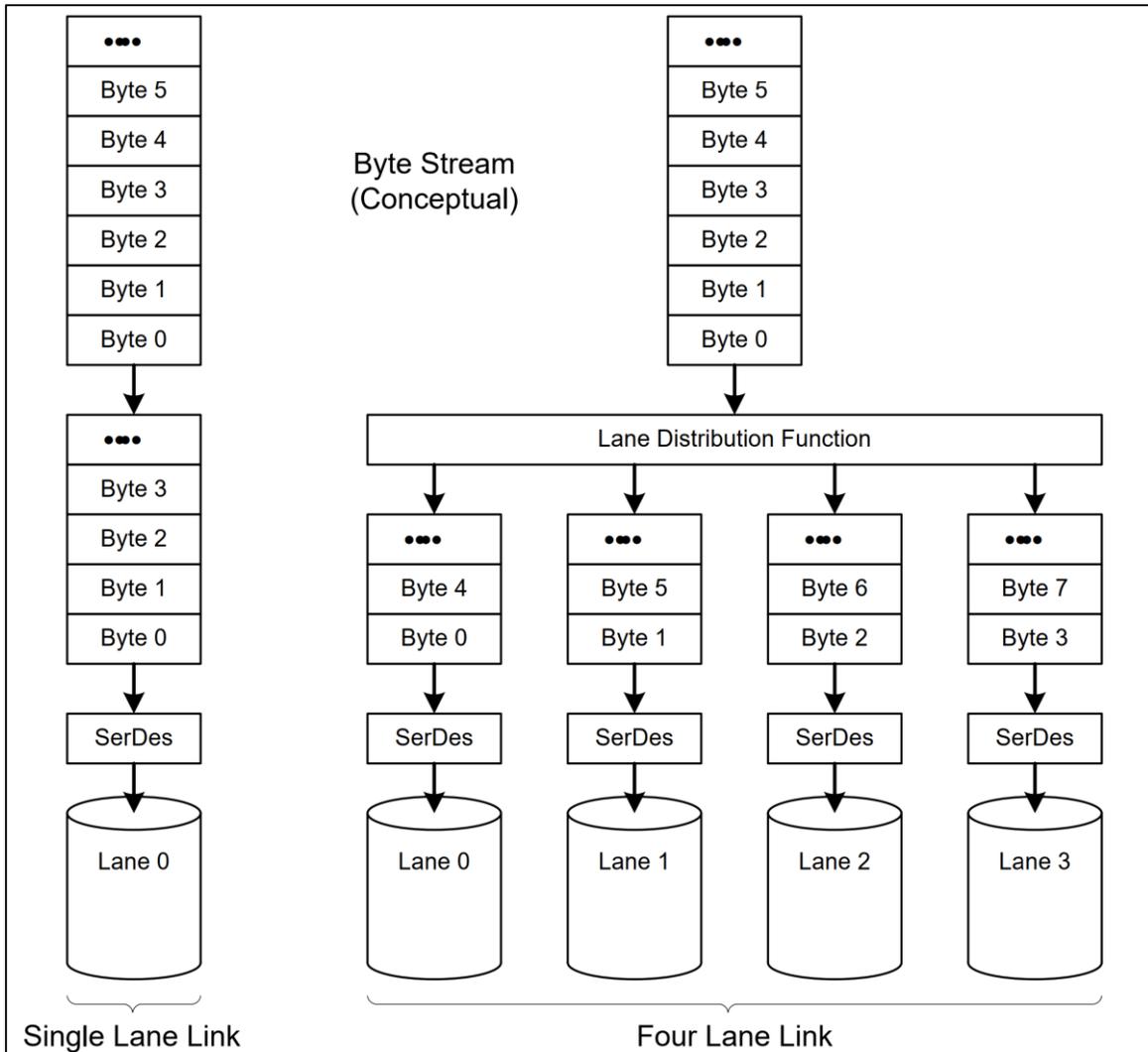


图 24.2.2.2 发送端通道管理层处理示意图

图 24.2.2.2 左侧是 1 Lane 的时候数据如何在数据线上传输, 由于只有 1 Lane, 所以上层传递下来的数据就只能按照 Byte0~ByteN 这样的顺序传输。

图 24.2.2.2 右侧是 4 lane 的时候数据传输方式, 由于采用了 4 通道, 那么上层传递下来的数据就要平均分配给 4 个通道。分配方法也很简单, 每个通道一个字节, 比如 Byte0 是 Lane0, Byte1 是 Lane1, Byte2 是 Lane2, Byte3 是 Lane3, 如此反复循环。

如果要发送的数据和通道数不是整数倍数, 那么先发送完的数据通道就进入 EOT(End of Transmission)模式。我们来看一个 2 Lane 整数倍传输和非整数倍传输的图, 如图 24.2.2.3 所示:

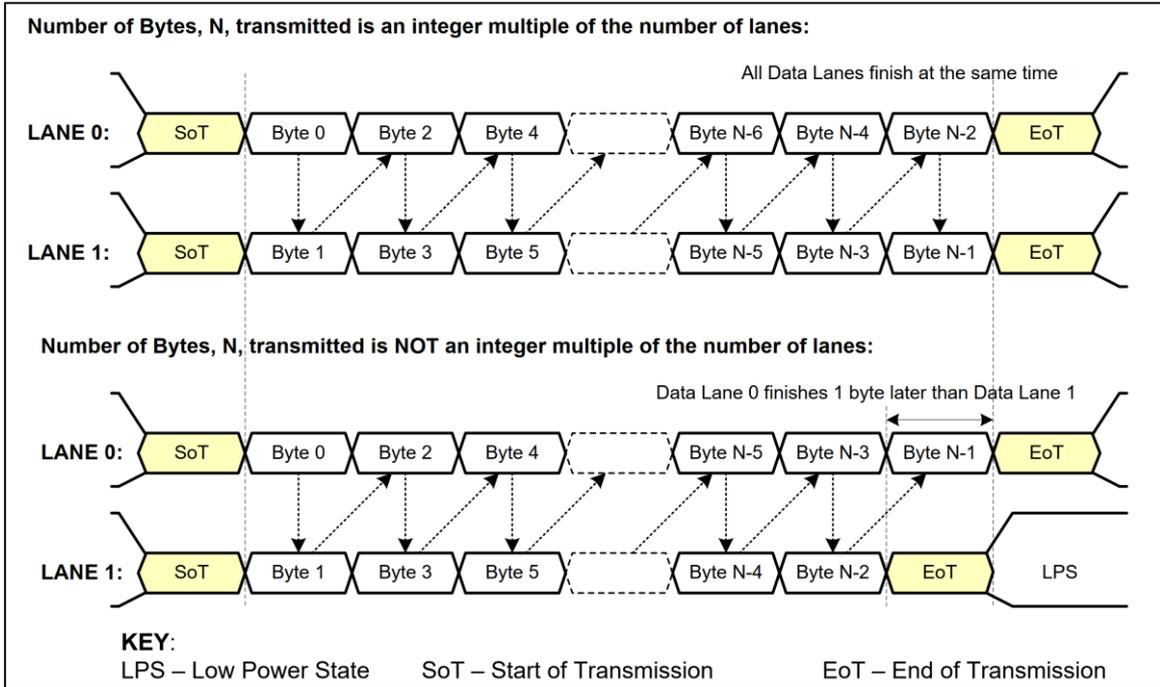


图 24.2.2.3 整数倍和非整数倍数据传输方式

图 24.2.2.3 中上部分就是整数倍传输，2 条通道一起结束，进入 EoT 模式。下图是非整数倍传输，其中 Lane 1 先传输完，所以 Lane 1 先进入 EoT 模式。同理，3/4 Lane 也一样。

在接收端执行相反的操作，将 Lane 上的数据整理打包成串行数据上报给上层，如图 24.2.2.4 所示：

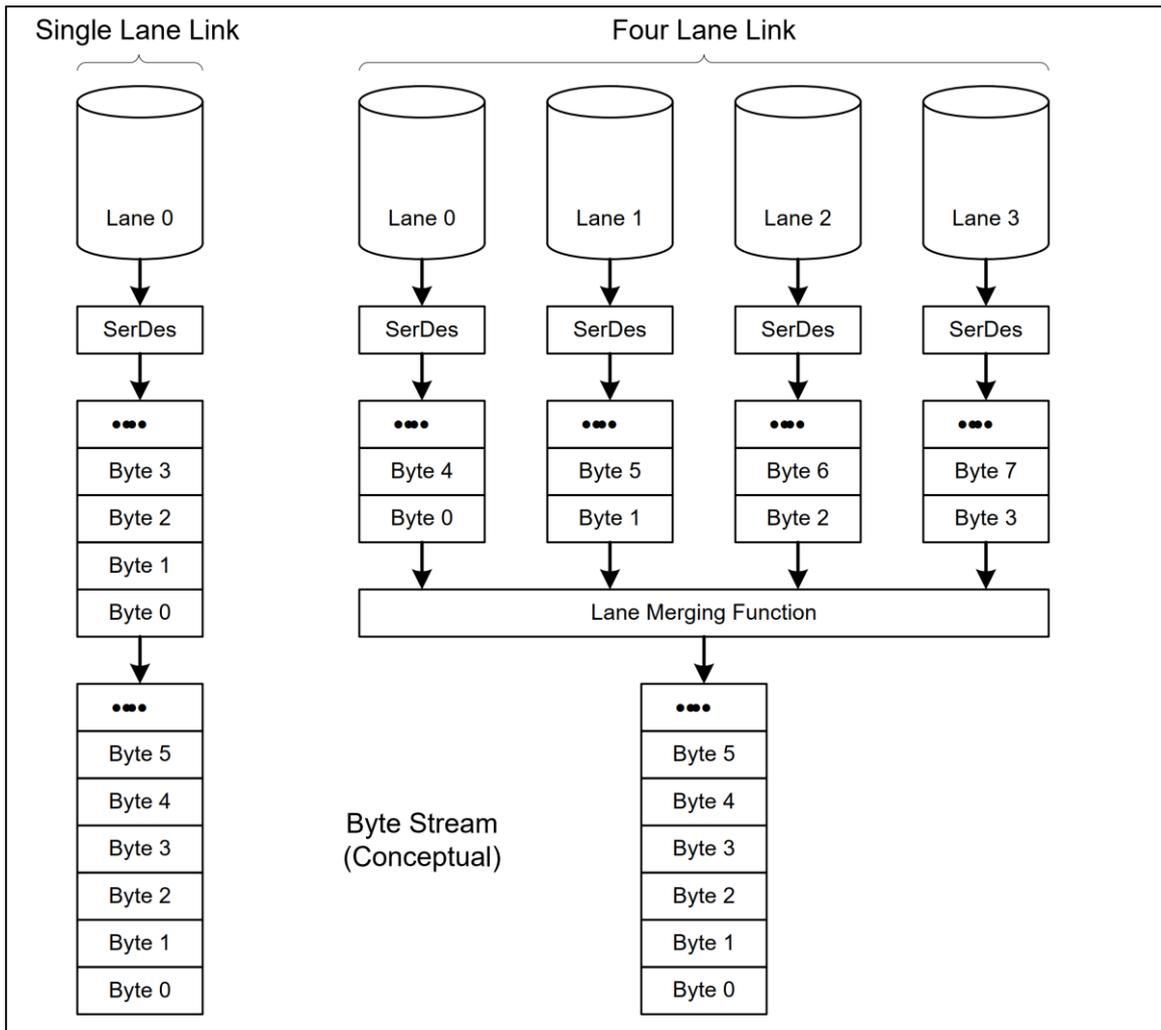


图 24.2.2.4 接收端通道管理层处理示意图

4、物理层

物理层就是最底层了，完成 MIPI DSI 数据在具体电路上的发送与接收，与物理层紧密相关的就是 D-PHY。物理层规定了 MIPI DSI 的整个电气属性，信号传输的时候电压等，关于物理层后面会详细讲解。

24.3 MIPI DSI 物理层和 D-PHY

MIPI DSI 的物理层也叫 PHY 层，前面说了 MIPI 有 C-PHY、D-PHY 等，只是在 MIPI DSI 领域 D-PHY 用的最多，所以这里可以简单的认为 MIPI DSI 物理层说的就是 D-PHY，本文后面统一用 D-PHY 来表示 MIPI DSI 的物理层。

D-PHY 是一个源同步、高速、低功耗、低开销的 PHY，特别适合移动领域。D-PHY 主要用于主处理器的摄像头和显示器外设，比如 MIPI 摄像头和屏幕。D-PHY 提供了一个同步通信接口来连接主机和外设，在实际使用中提供一对时钟线以及一对或多对信号线。时钟线是单向的，由主控产生，发送给设备。数据线根据实际配置，可以有 1~4 Lane，只有 Data0 这一组数据线可以是单向也可以是双向的，其他组的数据线都是单向的。

数据链路分为 High-Speed 模式和 Low-Power 模式，也就是常说的 HS 和 LP。HS 模式用来传输高速数据，比如屏幕像素数据。LP 模式用来传输低速的异步信号，一般是配置指令，屏幕

的配置参数就是用 LP 模式传输的。HS 模式下每个数据通道速率为 80~1500Mbps, 如果使用 deskew 校准, 可以到 2500Mbps, 最新版本的 D-PHY 支持的速率更高! LP 模式下最高 10Mbps。

24.3.1 什么是 Lane

我们前面已经提了很多次“Lane”这个词, 英文直译过来就是: 航道、跑道。在这里就是在主控与外设直接传输数据的通道, MIPI DSI 包括一个时钟 Lane 和多个数据 Lane, 每条 Lane 使用 2 根差分线来连接主控和外设。收发端都有对应的 Lane 模块来处理相关的数据, 一个完整的 Lane 模块如图 24.3.1.1 所示:

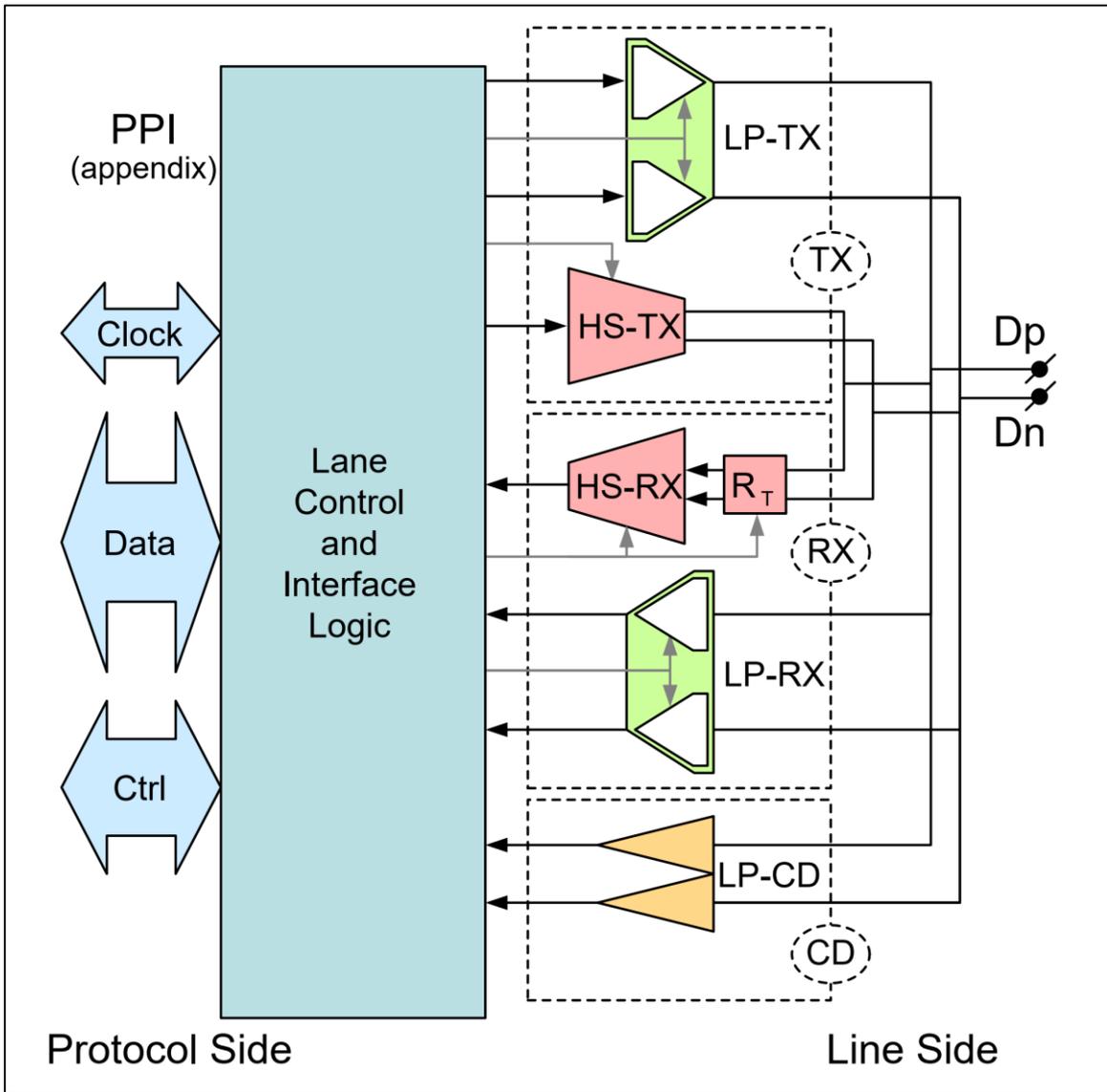


图 24.3.1.1 通用的 Lane 模块

从图 24.3.1.1 可以看出, 一个通用的 Lane 模块, 包括一个高速收发器和一个低速收发器, 其中高速收发器有 HS-TX、HS-RX, 低速收发器有 LP-RX 和 LP-TX, 以及一个低速竞争检测器 LP-CD。实际的 Lane 模块是在图 24.3.1.1 中简化而来的, 比如对于高速单向数据通道, 可能只有 HS-TX 或者 HS-RX。

24.3.2 D-PHY 信号电平

Lane 分为 HS 和 LP 两种模式，其中 HS 采用低压差分信号，传输速度快，但是功耗大，信号电压幅度 100mV~300mV，中心电平 200mV。LP 模式下采用单端驱动，功耗小，速率低 (<10Mbps)，信号电压幅度 0~1.2V。在 LP 模式下只使用 Lane0(也就是数据通道 0)，不需要时钟信号，通信过程的时钟信号通过 Lane0 两个差分线异或得到，而且是双向通信。HS 和 LP 模式下的信号电平如图 24.3.2.1 所示：

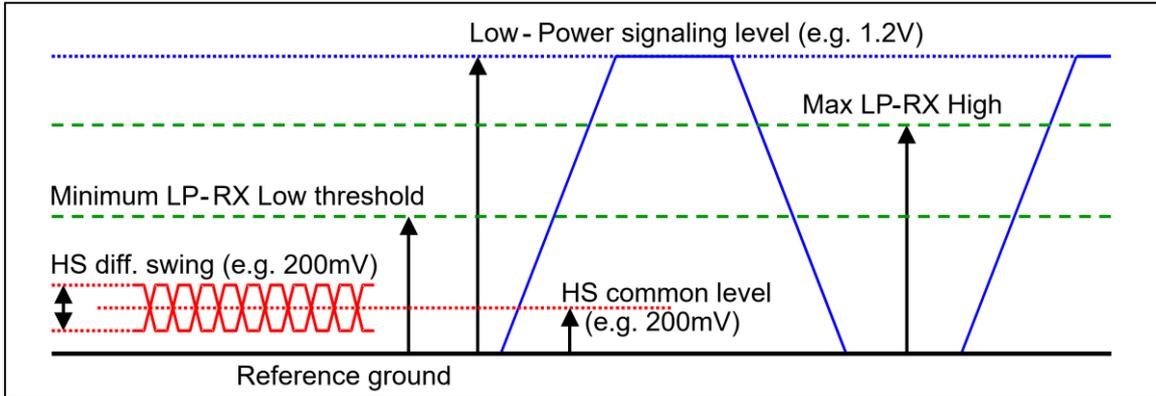


图 24.3.2.1 HS 和 LP 模式信号电平

图 24.3.2.1 中蓝色实线是 LP 模式下的信号波形示例，电压为 0~1.2V。绿色虚线是 LP 模式下信号的高低电平门限。红色实线是 HS 模式下的信号波形示例，中心电平 200mV。

24.3.3 通道状态

上面说了 HS 模式下是单向差分信号，主控发送(HS_TX)，外设接收(HS_RX)。而 LP 是双向单端信号，接收和发送端都有 LP_TX 和 LP_RX，注意只有 Lane0 能做 LP。

由于 HS 采用差分信号，所以只有两种状态：

HS-0: 高速模式下 Dp 信号低电平，Dn 信号高电平的时候。

HS-1: 高速模式下 Dp 信号高电平，Dn 信号低电平的时候。

LP 模式下有两根独立的信号线驱动，所以有 4 个状态：

LP-00: 后面的“00”对应两根信号线的电平状态，第 1 个 0 表示 Dp 为低电平，第 2 个 0 表示 Dn 为低电平。如果是高电平，那么就为 1。因此 LP 模式剩下的三个状态就是 LP-01、LP-10 和 LP-11。

这 6 种状态对应的功能如图 24.3.3.1 所示：

State Code	Line Voltage Levels		High-Speed	Low-Power	
	Dp-Line	Dn-Line	Burst Mode	Control Mode	Escape Mode
HS-0	HS Low	HS High	Differential-0	N/A, Note 1	N/A, Note 1
HS-1	HS High	HS Low	Differential-1	N/A, Note 1	N/A, Note 1
LP-00	LP Low	LP Low	N/A	Bridge	Space
LP-01	LP Low	LP High	N/A	HS-Rqst	Mark-0
LP-10	LP High	LP Low	N/A	LP-Rqst	Mark-1
LP-11	LP High	LP High	N/A	Stop	N/A, Note 2

图 24.3.3.1 6 种状态

通过图 24.3.3.1 种这 6 个状态的转换，D-PHY 就能工作在不同的工作模式。

24.3.4 数据 Lane 三种工作模式

D-PHY 协议规定了，通过 Lane 的不同状态转换有三种工作模式：控制模式、高速模式和 Escape 模式。控制模式和 Escape 模式都属于 LP，高速模式属于 HS。正常情况下，数据 Lane 工作在控制模式或者高速模式下，

1、高速模式

高速模式用于传输实际的屏幕像素数据，采用突发(Bursts)传输方式。为了帮助接收端同步，需要在数据头尾添加一些序列，接收端在接收到数据以后要把头尾去掉。高速数据传输起始于 STOP 状态(LP-11)，并且也中终于 STOP 状态(LP-11)。在高速模式下传输数据的时候始终 Lane 也工作与 HS 模式，提供 DDR 时钟，也就是双边沿时钟，在时钟频率不变的情况下，传输速率提高一倍，这样可以有效利用带宽。

当数据传输请求发出以后，数据 Lane 退出 STOP 模式进入到高速模式，顺序是：LP-11→LP-01→LP-00。然后发出一个 SOT 序列(Start-of-Transmission)，SOT 后面跟着的就是实际的负载数据。当负载数据传输结束以后会紧跟一个 EOT 序列(End-of-Transmission)序列，数据线直接进入进入到 STOP 模式。图 24.3.4.1 是一个基础的高速传输结构示意图：

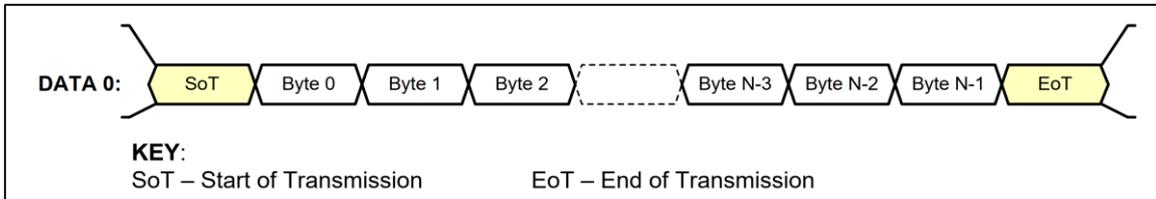


图 24.3.4.1 基础的高速数据传说结构体

图 24.3.4.1 只是展示了一条 Lane，负载数据前面是一个 SoT，传输完成以后紧跟一个 EoT，中间就是实际的负载数据。

一个完整的高速模式数据传输时序如图 24.3.4.2 所示：

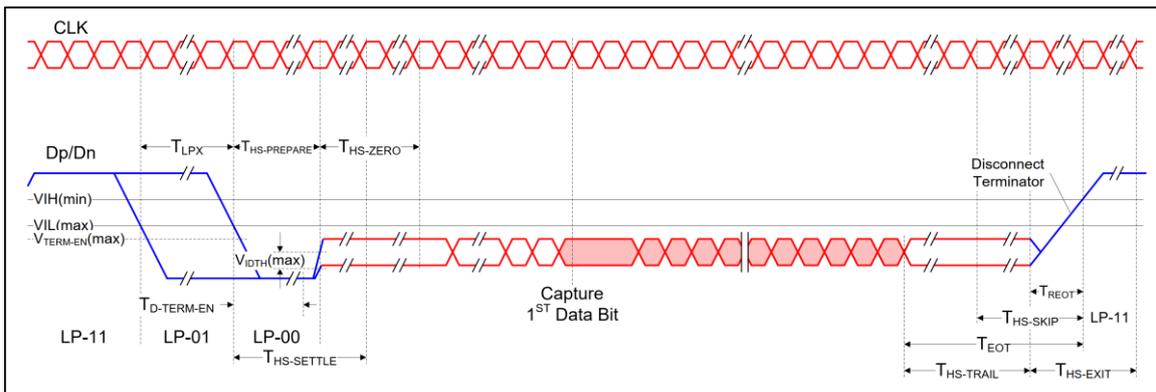


图 24.3.4.2 高速数据传输时序

图 24.3.4.2 中左侧蓝色部分是进入 HS 模式，要从 LP-11→LP01→LP-00，然后数据线进入到 HS 模式，也就是中间红色部分，传输实际的数据。传输完成以后重新进入到 LP-11(STOP)模式，也就是右边的蓝色部分。

2、Escape 模式

Escape 是运行在 LP 状态下的一个特殊模式，在此模式下可以实现一些特殊的功能，我们给屏幕发送配置信息就需要运行在 Escape 模式下。数据线进入 Escape 模式的方式为：LP-11→

LP-10→LP-00→LP-01→LP-00。退出 Escape 模式的方式为：LP-00→LP-10→LP-11，也就是最后会进入到 STOP 模式，进入和退出 Escape 模式的时序如图 24.3.4.3 所示：

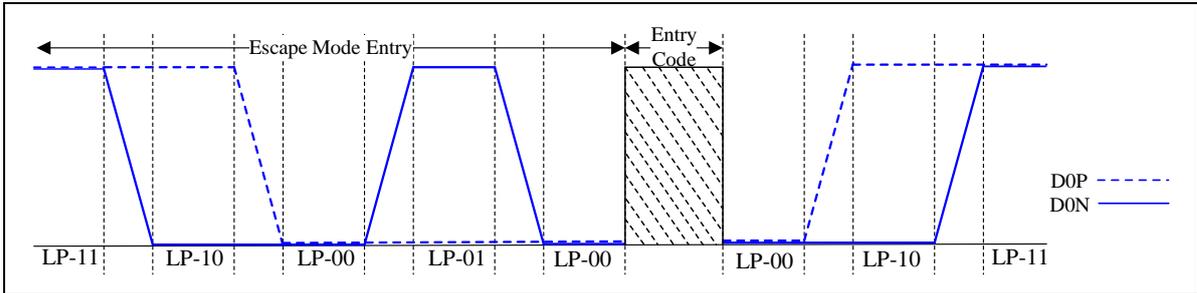


图 24.3.4.3 Escape 模式进入和退出

对于数据 Lanes，进入 Escape 模式以后，应该紧接着发送一个 8bit 的命令来表示接下来要做的操作，命令如图 24.3.4.3 所示：

Escape Mode Action	Command Type	Entry Command Pattern (first bit transmitted to last bit transmitted)
Low-Power Data Transmission	mode	11100001
Ultra-Low Power State	mode	00011110
Undefined-1	mode	10011111
Undefined-2	mode	11011110
Reset-Trigger [Remote Application]	Trigger	01100010
Unknown-3	Trigger	01011101
Unknown-4	Trigger	00100001
Unknown-5	Trigger	10100000

图 24.3.4.3 Escape 模式命令

从图 24.3.4.3 可以看出，有三个可选的命令：LPDT(Low-Power Data Transmission)、ULPS(Ultra-Low Power State)和 Remote-Trigger(这里没有写错，因为这个命令大部分做复位操作，所以有些资料将这个命令也叫做 Remote-Trigger)。

1)、LPDT 命令

图 24.3.4.3 中第一个就是 LPDT 命令，命令序列为 11100001，注意低 bit 先发送，所以对应的十六进制就是 0X87(0X10000111)！LPDT 直译过来就是低功耗数据传输，我们在初始化 MIPI 屏幕的时候发送的初始化序列就需要用 LPDT 命令，后面会给大家看逻辑分析仪采集到的实际数据。LPDT 命令序列后面紧跟着就是要发送的数据，分为长包和短包两种，长短包结构后面会详细讲解。

2)、ULPS 命令

ULPS 命令是让 Lane 进入超低功耗模式。

3)、Remote-Trigger 命令

注意，这里大家可能会疑惑，有的资料叫做 Remote-Trigger，有的叫做 Reset-Trigger，其实都是一个东西。因为本质是 Remote Application，但是做的是 Reset 的工作，所以就产生了两种叫法，目前此命令就是用于远程复位。

Escape 模式下发送这三个命令的时序图如图 24.3.4.4 所示：

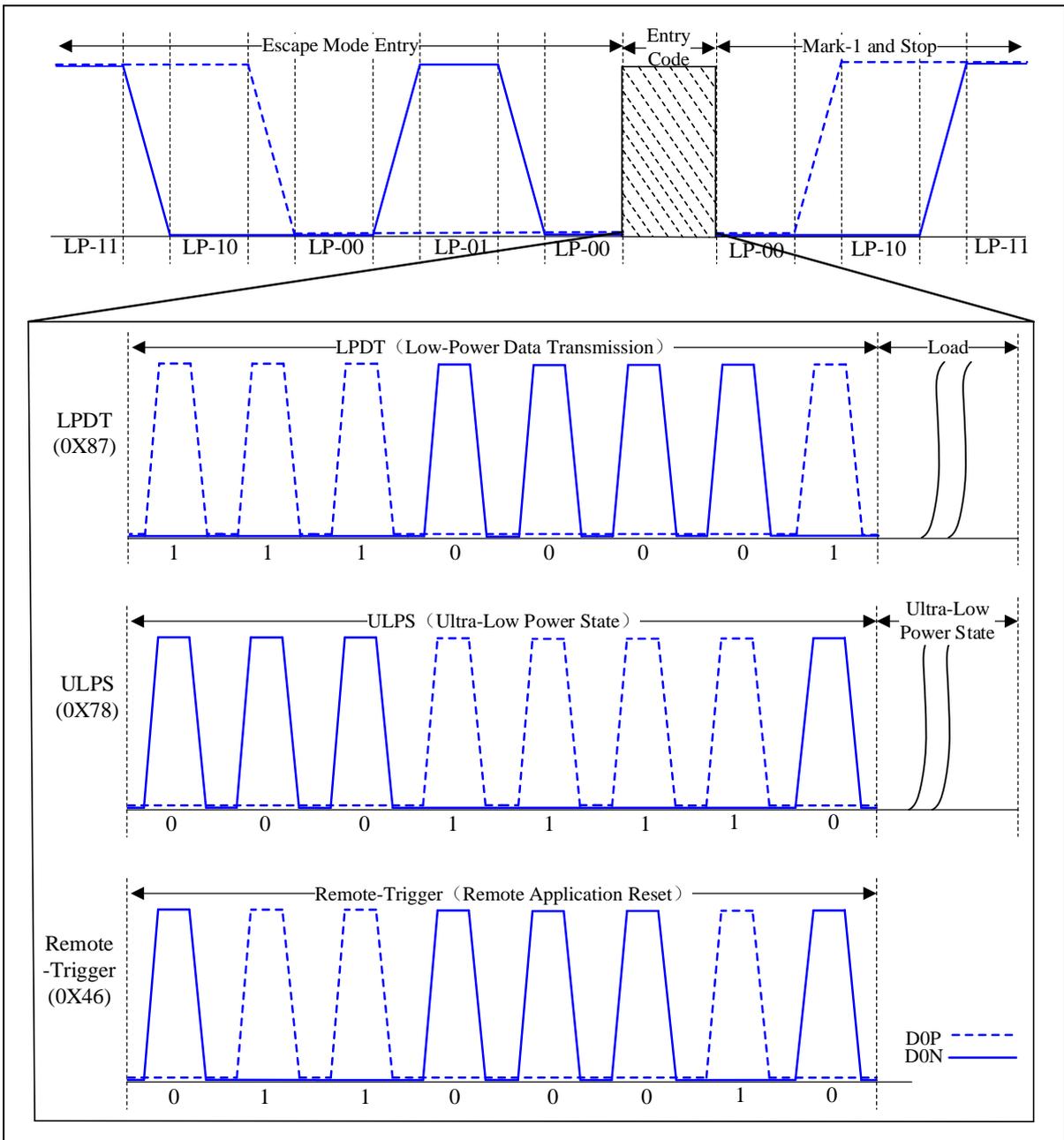


图 24.3.4.4 Escape 模式下各命令时序图

24.5 video 和 command 模式

在 MIPI DSI 的链路层有两种模式: video(视频)和 command(命令)模式, 这个属于 HOST 端, 也就是主控端, 比如 RK3568 的 DSI HOST 接口。video 和 command 通常离不开 HS 和 LP 模式, 但是 video 和 command 属于 Host 范畴, HS 和 LP 属于 D-PHY 范畴。

24.5.1 command 模式

command 模式一般是针对那些含有 buffer 的 MCU 屏幕, 比如 STM32 单片机驱动 MCU 屏的时候就是 command 模式。当画面有变化的时候, DSI Host 端将数据发给屏幕, 主控只有在画面需要更改的时候发送像素数据, 画面不变化的时候屏幕驱动芯片从自己内部 buffer 里面提取

数据显示, command 模式下需要双向数据接口。一般此种模式的屏幕尺寸和分辨率不大, 一般用在单片机等低端领域。command 模式如图 24.5.1.1 所示:

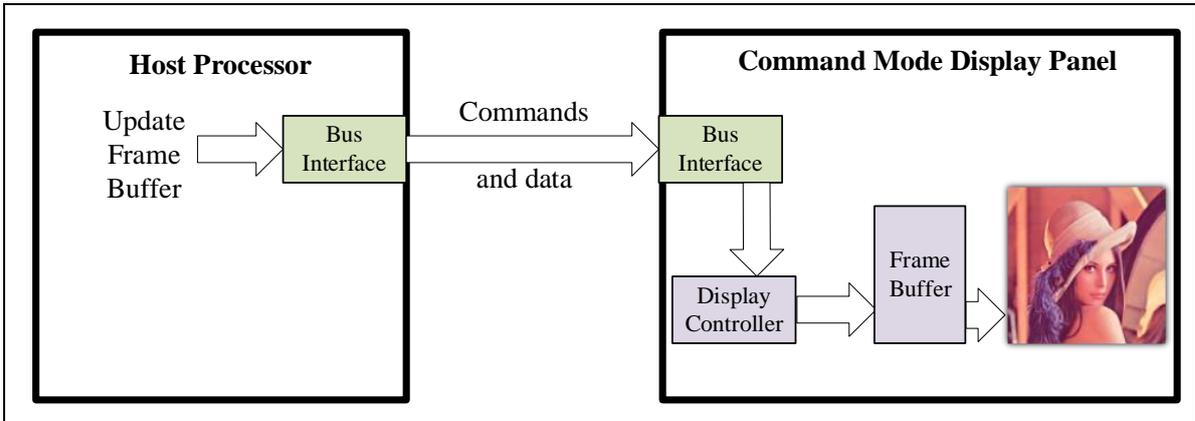


图 24.5.1.1 command 模式示意图

24.5.2 video 模式

video 模式没有 framebuffer, 需要主控一直发送数据给屏幕, 和我们使用过的 RGB 接口屏幕类似。但是 MIPI DSI 没有专用的信号线发送同步信息, 比如 VSYNC、HSYNC 等, 所以这些控制信号和 RGB 图像数据以报文的形式在 MIPI 数据线上传输。基本上我们说的“MIPI 屏”就是工作在 video 模式下, 包括我们使用的 RK3568, 其工作模式就是 video。video 模式如图 24.5.1.2 所示:

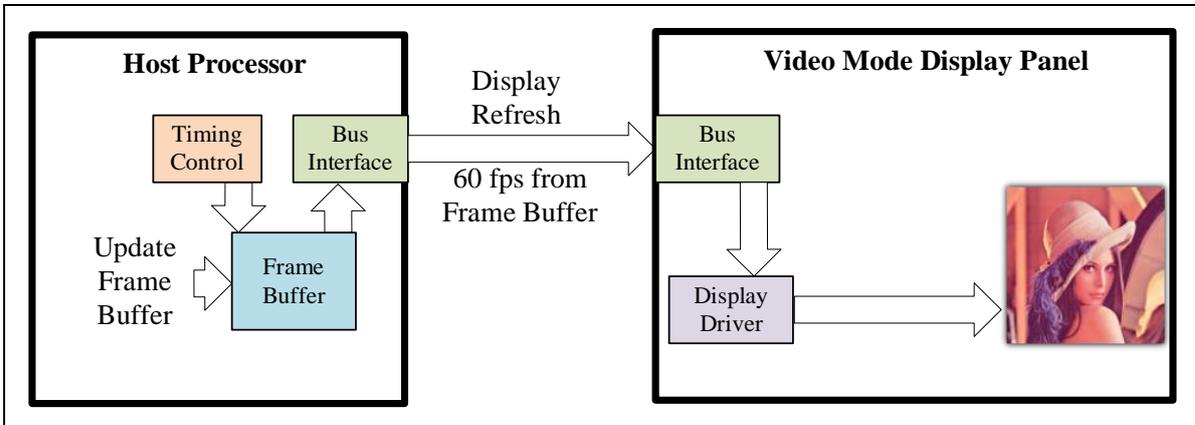


图 24.5.1.2 video 模式示意图

接下来是重点:

笔者在学习 command 和 video 模式的时候网上很多资料都说: **command 模式下物理层可以使 HS, 也可以是 LP。但是 video 模式下只能是 HS!** 但是笔者在使用 RK3568 驱动 MIPI 屏幕的时候明明是工作在 video 模式下的, 但是 HS 和 LP 都支持, 和网上的资料相悖, 困惑了很久。最后笔者在 RK3568 官方手册里面找到了答案:

对于 RK3568 的 MIPI DSI HOST 控制器而言, 在 video 模式下, 支持 HS 和 LP 下发送命令, 当, DSI 控制器使用 BLLP(Blanking or Low-Power periods)来传输命令以取代通过 APB 通用接口。所以对于 RK3568 而言, 虽然它工作在 video 模式下, 但是依旧可以使用 LP 来传输配置信息给屏幕。关于 BLPP 更加详细的讲解, 自行查阅 RK3568 参考手册的 MIPI DSI HOST 控制器章节。

以上为笔者查找资料总结的结论，如哪位大神发现错误还请留言，指出错误所在。

24.6 DBI 和 DPI 格式

关于 DBI 和 DPI 这两种格式的详细协议内容，请参考《MIPI Alliance Standard for Display Bus Interface (V2.0) .pdf》和《MIPI Alliance Standard for Display Pixel Interface (DPI-2) .pdf》这两份文档。

我们首先先了解几个名词，如表 24.6.1 所示：

缩写	英文全称	含义
HSYNC	Horizontal Sync	水平同步
HLW/HPW	Horizontal Low Pulse Width	水平同步信号宽度
HSA	Horizontal Sync Active	水平同步有效
HSS	Horizontal Sync Start	水平同步开始
HSE	Horizontal Sync End	水平同步结束
HBP	Horizontal Back Porch	水平后肩
HFP	Horizontal Front Porch	水平前肩
HACT	Horizontal Active	水平有效区域，也就是屏幕有效宽度
VSYNC	Vertical Sync	垂直同步
VLW/VPW	Vertical Low Pulse Width	垂直同步信号宽度
VSA	Vertical Sync Active	垂直同步信号宽度
VSS	Vertical Sync Start	垂直同步开始
VSE	Vertical Sync End	垂直同步结束
HBP	Vertical Back Porch	垂直后肩
VACT	Vertical Active	垂直有效区域，也就是屏幕有效高度
VFP	Vertical Front Porch	垂直前肩
RGB	—	在这里指 RGB 原始像素数据流
LPM	Low Power Mode	低功耗模式
BLLP	Blanking or Low-Power periods	没有包含有效数据的数据包或者进入 LP 模式下的状态，称为 BLLP

表 24.6.1 相关名词解释表

其中垂直同步 VSYNC 表示一帧图像的起始，水平同步 HSYNC 表示一行图像的起始。

24.6.1 DBI 接口

DBI 接口全称是 Display Bus Interface，俗称 MCU 接口、8080 接口。也就是大家在用 STM32F103/407 这种 MCU 的时候使用的屏幕接口。MCU 通过并行接口传输控制命令和数据，DBI 接口示意图如图 24.6.1.1 所示：

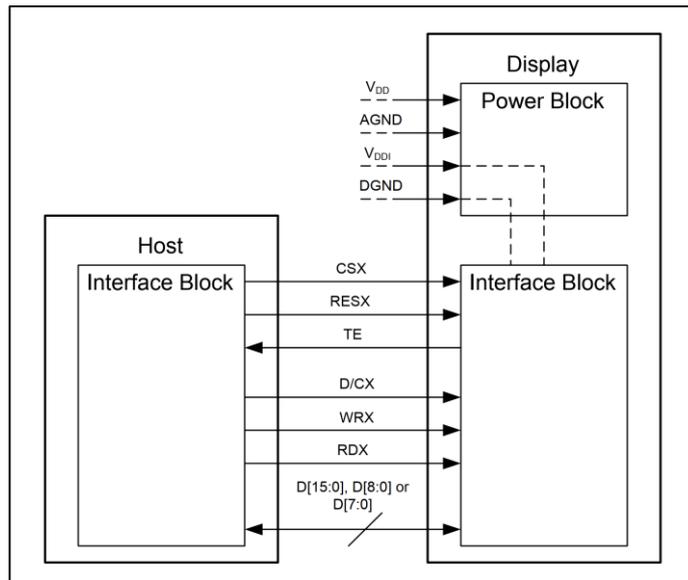


图 24.6.1.1 DBI 接口示意图

正点原子 STM32F103 和 F407 开发板的屏幕就是用的 DBI 接口，也就是所谓的 MCU 口，正点原子 4.3 寸 MCU 屏的 TFT 接口部分原理图如图 24.6.1.2 所示：

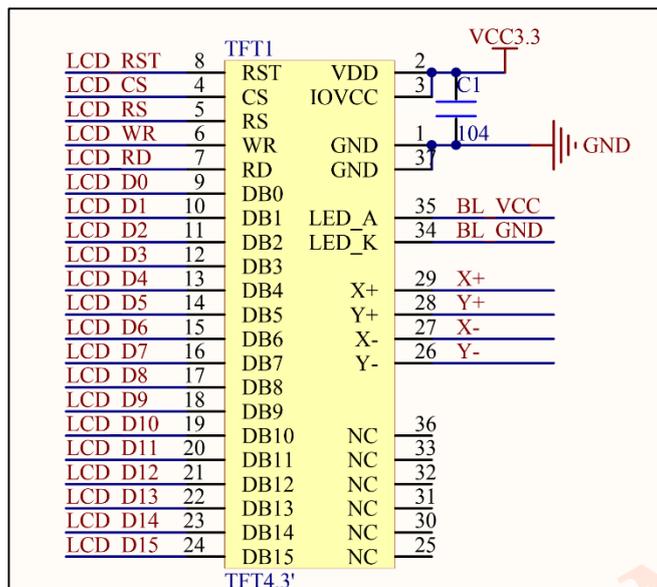


图 24.6.1.2 4.3 寸 MCU 屏 TFT 接口

对于 DBI 接口屏幕而言，有 CS/RS/WR/RD 控制线，以及 D0~D15 数据线，DBI 是非常低端的产品所使用的接口，比如低端 MCU 等。

24.6.2 DPI 接口

DPI 接口全称 Display Pixel Interface，就是我们常说的 RGB 接口，RGB 接口使用场合非常多，比如 STM32H7 单片机，大量的 Cortex-A 系列内核的 MPU，包括 RK3568 本身就支持 RGB 接口，DPI 是目前很多中低端芯片的首选屏幕接口。

1、LCD 时间参数

以一个 720*1280 分辨率的 LCD 为例，其显示结构如图 24.6.2.1 所示：

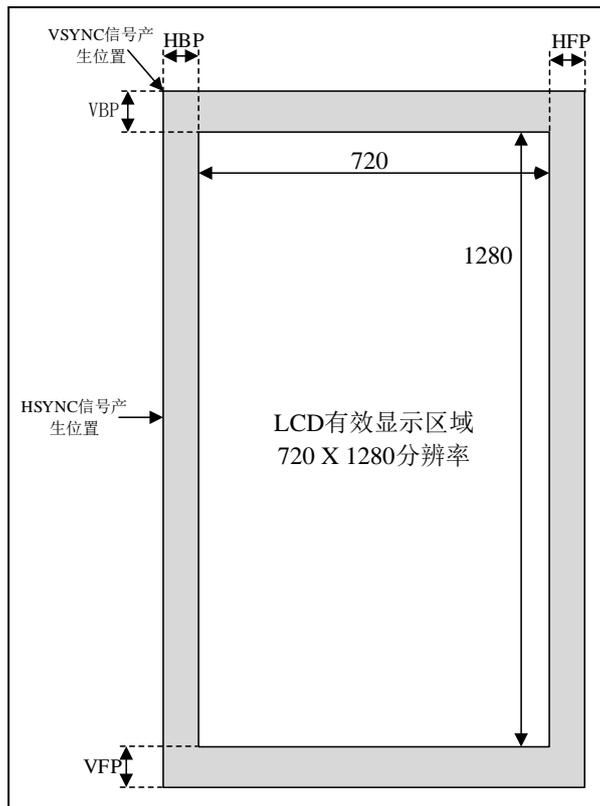


图 24.6.2.1 LCD 显示示意图

结合图 24.6.2.1 我们来看一下 LCD 是怎么扫描显示一帧图像的。一帧图像也是由一行一行组成的。HSYNC 是水平同步信号，也叫做行同步信号，当产生此信号的话就表示开始显示新的一行了，所以此信号都是在图 24.6.2.1 的最左边。当 VSYNC 信号是垂直同步信号，也叫做帧同步信号，当产生此信号的话就表示开始显示新的一帧图像了，所以此信号在图 24.6.2.1 的左上角。

在图 24.6.2.1 可以看到有一圈“黑边”，真正有效的显示区域是中间的白色部分。那这一圈“黑边”是什么呢？这就要从显示器的“祖先”CRT 显示器开始说起了，CRT 显示器就是以前很常见的那种大屁股显示器，在 2023 年应该很少见了，如果在农村应该还是可以见到的。CRT 显示器屁股后面是个电子枪，这个电子枪就是我们上面说的“画笔”，电子枪打出的电子撞击到屏幕上的荧光物质使其发光。只要控制电子枪从左到右扫完一行(也就是扫描一行)，然后从上到下扫描完所有行，这样一帧图像就显示出来了。也就是说，显示一帧图像电子枪是按照‘Z’形在运动，当扫描速度很快的时候看起来就是一幅完成的画面了。

当显示完一行以后会发出 HSYNC 信号，此时电子枪就会关闭，然后迅速的移动到屏幕的左边，当 HSYNC 信号结束以后就可以显示新的一行数据了，电子枪就会重新打开。在 HSYNC 信号结束到电子枪重新打开之间会插入一段延时，这段延时就图 24.6.2.1 中的 HBP。当显示完一行以后就会关闭电子枪等待 HSYNC 信号产生，关闭电子枪到 HSYNC 信号产生之间会插入一段延时，这段延时就是图 24.6.2.1 中的 HFP 信号。同理，当显示完一帧图像以后电子枪也会关闭，然后等到 VSYNC 信号产生，期间也会加入一段延时，这段延时就是图 24.6.2.1 中的 VBP。VSYNC 信号产生，电子枪移动到左上角，当 VSYNC 信号结束以后电子枪重新打开，中间也会加入一段延时，这段延时就是图 24.6.2.1 中的 VFP。

HBP、HFP、VBP 和 VFP 就是导致图 24.6.2.1 中黑边的原因，但是这是 CRT 显示器存在黑

边的原因, 现在是 LCD 显示器, 不需要电子枪了, 那么为何还会有黑边呢? 这是因为 RGB LCD 屏幕内部是有一个 IC 的, 发送一行或者一帧数据给 IC, IC 是需要反应时间的。通过这段反应时间可以让 IC 识别到一行数据扫描完了, 要换行了, 或者一帧图像扫描完了, 要开始下一帧图像显示了。因此, 在 LCD 屏幕中继续存在 HBP、HFP、VPB 和 VFP 这四个参数的主要目的是为了锁定有效的像素数据。这四个时间是 LCD 重要的时间参数, 在 DPI 接口的屏幕驱动中, 我们重点就是设置这几个时序参数。

2、DPI 接口连接

主控和 LCD 外设的 DPI 接口示意图如图 24.6.2.2 所示:

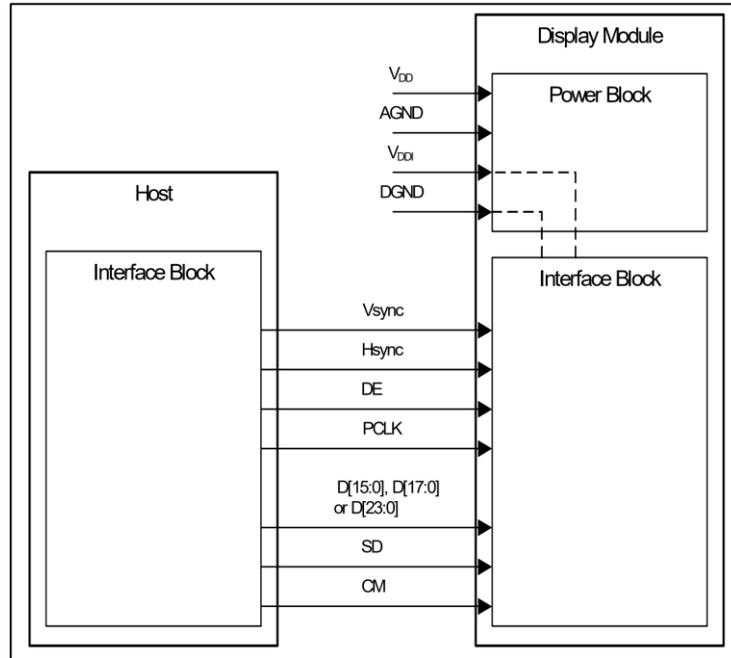


图 24.6.2.2 DPI 接口示意图

对于 DPI 接口, 也就是 RGB 屏, 一般有 DE、VSYNC、HSYNC、CLK 这几个控制线, 以及 D0~D23 数据线(如果采用 RGB888 格式的话)。正点原子的 7 寸 RGB 屏幕接口原理图如图 24.6.2.3 所示:

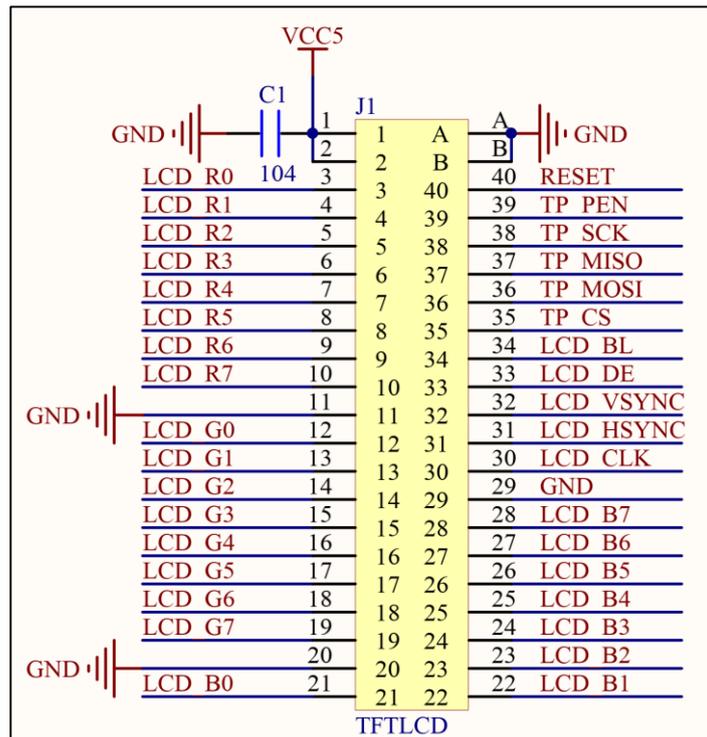


图 24.6.2.3 7 寸 RGB 屏幕接口原理图

重点知识:

为什么要花这么久时间给大家讲解 DPI 接口? 因为我们在 MIPI DSI 接口的屏幕里面传输的就是 DPI 格式的数据, 包括 HBP、HFP、VBP、VFP 等时序参数也是需要再 MIPI DSI 屏幕里面使用的!

24.7 video 模式下三种传输方式

对于 video 模式下的数据传输有三种时序模式:

●Non-Burst Mode with Sync Pulses: 外设可以准确的重建原始的视频时序, 包括同步脉冲宽度。

●Non-Burst Mode with Sync Events: 和上面的模式类似, 但是不需要精准的重建同步脉冲宽度, 取而代之的是发送一个“Sync event”包。

●Burst Mode: 此模式下发送 RGB 数据包的时间被压缩, 这样可以在发送一行数据以后尽快进入到 LP 模式, 以节省功耗。

在详细看这三个模式之前, 我们先了解几个名词, 如表 24.6.1 所示:

24.7.1 Non-Burst Mode with Sync Pluses

此模式下外设可以准确的重建原始视频的时序、同步信号。通过在 DSI 接口上发送 DPI 时序, 可以精确的匹配 DPI 像素传输速率以及时序宽度等, 比如同步脉冲。所以此模式下每一个 Sync Start(HSA)都要有一个对应的 Sync End(HSE), 此模式下的时序图如图 24.7.1 所示:

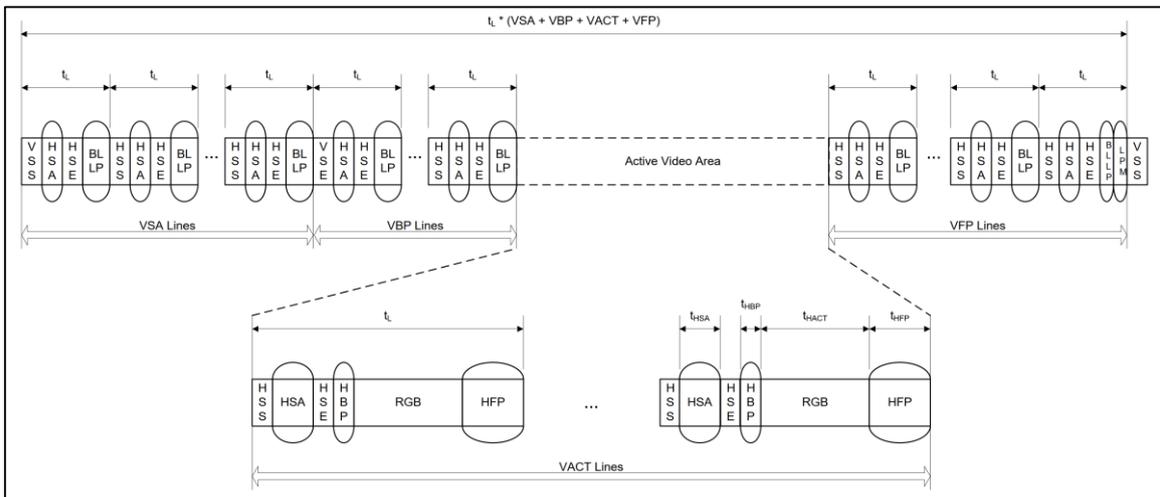


图 24.7.1.1 Non-Burst Mode with Sync Pluses 模式时序图

Non-Burst Mode with Sync Pluses 模式需要精准控制同步时序宽度, 所以有 Sync Start 和 Sync End, 比如图 24.7.1.1 中的 VSS 和 VSE、HSS 和 HSE 信号, 也就是水平同步开始和结束信号。

24.7.2 Non-Burst Mode with Sync Events

此模式和上一小节讲解的 Non-Burst Mode with Sync Pluses 模式类似, 只是此模式下不需要精准的控制同步时序的宽度, 所以此模式只有 Sync Start, 时序如图 24.7.1.2 所示:

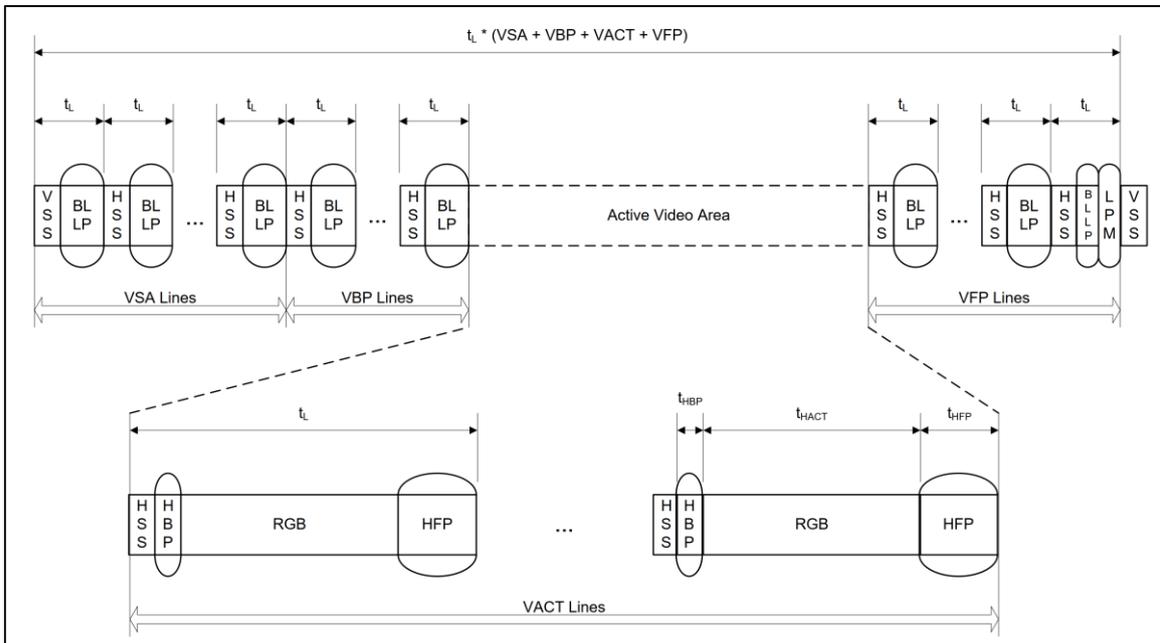


图 24.7.2.1 Non-Burst Mode with Sync Events 模式时序图

从图 24.7.1.2 可以看出此模式下只有 Sync Start, 没有 Sync End, 比如只有 VSS 和 HSS, 而没有 VSE 和 HSE。

24.7.3 Burst Mode

Burst Mode 是 MIPI DSI 最常用的模式，相比于 Non-Burst Mode with Sync Events 模式，Burst Mode 可以快速的完成一帧或一行图像像素的传输，这样可以使数据线有更多的时间处于 LP 模式以节省功耗。此模式下时序图如图 24.7.3.1 所示：

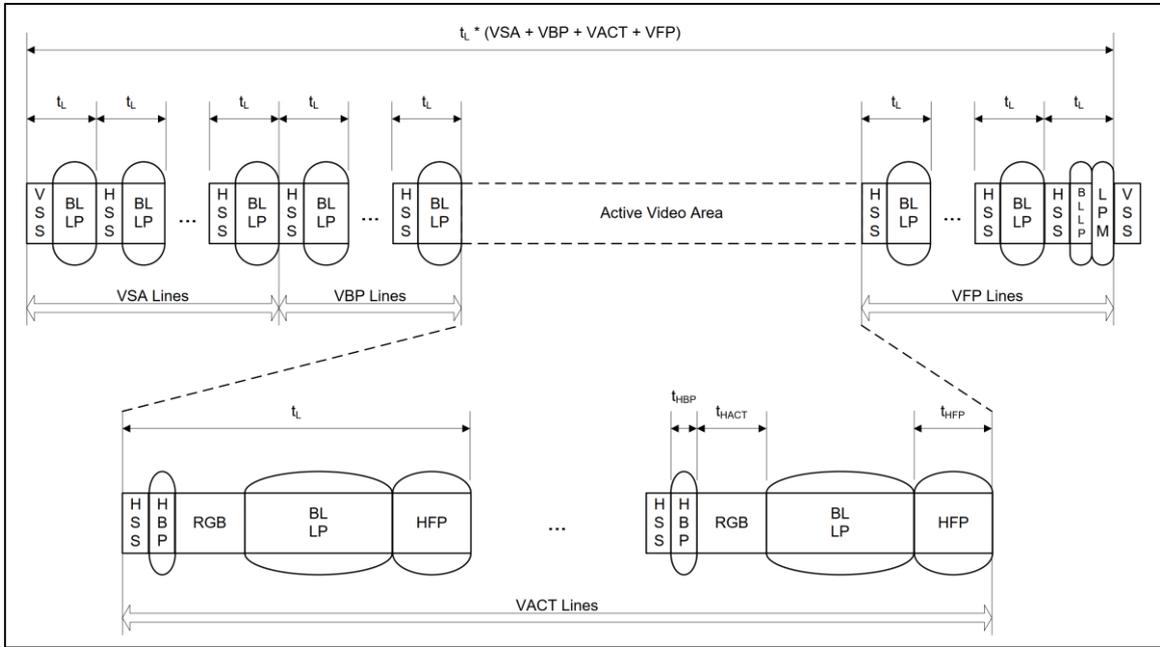


图 24.7.3.1 Burst Mode 模式时序图

图 24.7.3.1 中 VACT Lines 中，RGB 图像像素数据会尽快传递完成，然后进入到 BLLP 以节省功耗。

24.8 长短数据包

24.8.1 数据包概述

在 MIPI DSI 的数据传输中，不管是并行数据、信号事件还是命令，都需要按照协议打包成数据包。按照规定的协议添加头尾等信息，然后通过数据 Lane 将打包好的数据发送出去。

如一次传输只发送一个数据包，那么在传输多个数据包就会花费大量的开销在 LPS 和 HS 切换上，这样会严重的浪费带宽。为此，MIPI DSI 协议允许在一次传输中可以串行的发送多个数据包，这样就可以大幅的提高带宽利用率，这对于像外设初始化这种操作非常有益，比如我们在初始化屏幕的时候会发送大量的初始化命令。

数据包第一个字节是 DI(Data Identifier)，用来指定当前数据包的含义，数据包一共有两种数据包：

- **短数据包：**固定 4 个字节，包括 ECC。短包一般用于 Command 模式下发送命令和参数，但是在实际使用中，Video 模式下也用短包发送命令参数信息。其他的一些短包发送一些事件，比如 H Sync 和 V Sync 边沿。

- **长数据包：**通过 2 个字节的 WC(Word Count)域来指定负载长度，负载长度范围：0~2¹⁶-1 个字节，也就是 0~65535 个字节。因此一个长数据包最多有 65541 个字节：

1 个字节的 DI+2 个字节的 WC+1 个字节的 ECC+65535 个字节的负载+2 个字节的校验和 =1+2+1+65535+2=65541。

我们在 24.3.4 小节讲解 Escape 模式的时候说过，进入 Escape 模式以后紧跟着一个 8bit 的命令来表示接下来要做的操作，我们一般使用 LPDT 命令+具体的配置参数来完成向 MIPI 屏幕发送初始化参数的操作。但是并不是直接在 LPDT 命令后面跟着发送想要的时序参数就行了，而是要按照上面说的 MIPI DSI 格式将时序参数打包成长短数据包发送出去。

24.8.2 数据包字节排序策略

数据包都是以字节形式通过接口传输，同一个字节 LSB 先传输，MSB 后传输，也就是低 bit 先发，高 bit 后发。如果某个域有多个字节，那么低字节(LS Byte)先发，高字节(MS Byte)后发，比如 WC 字段有 2 个字节，那么低字节的就先发，高字节的后发。但是，对于负载段就不用按照这个规定来，而是按照字节顺序发送。

图 24.8.2.1 就是一个长包数据的发送方式：

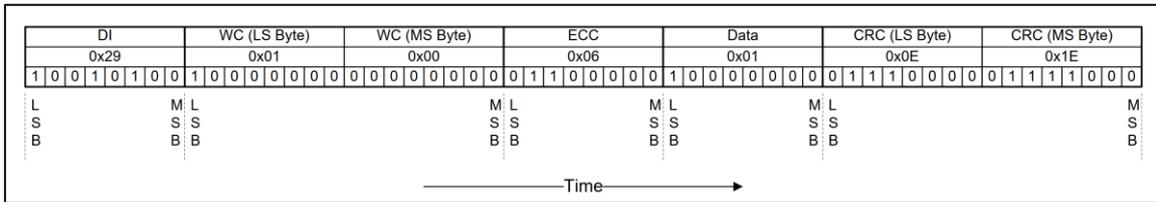


图 24.8.2.1 长包数据发送格式

24.8.3 长数据包

长数据包结构如图 24.8.3.1 所示：

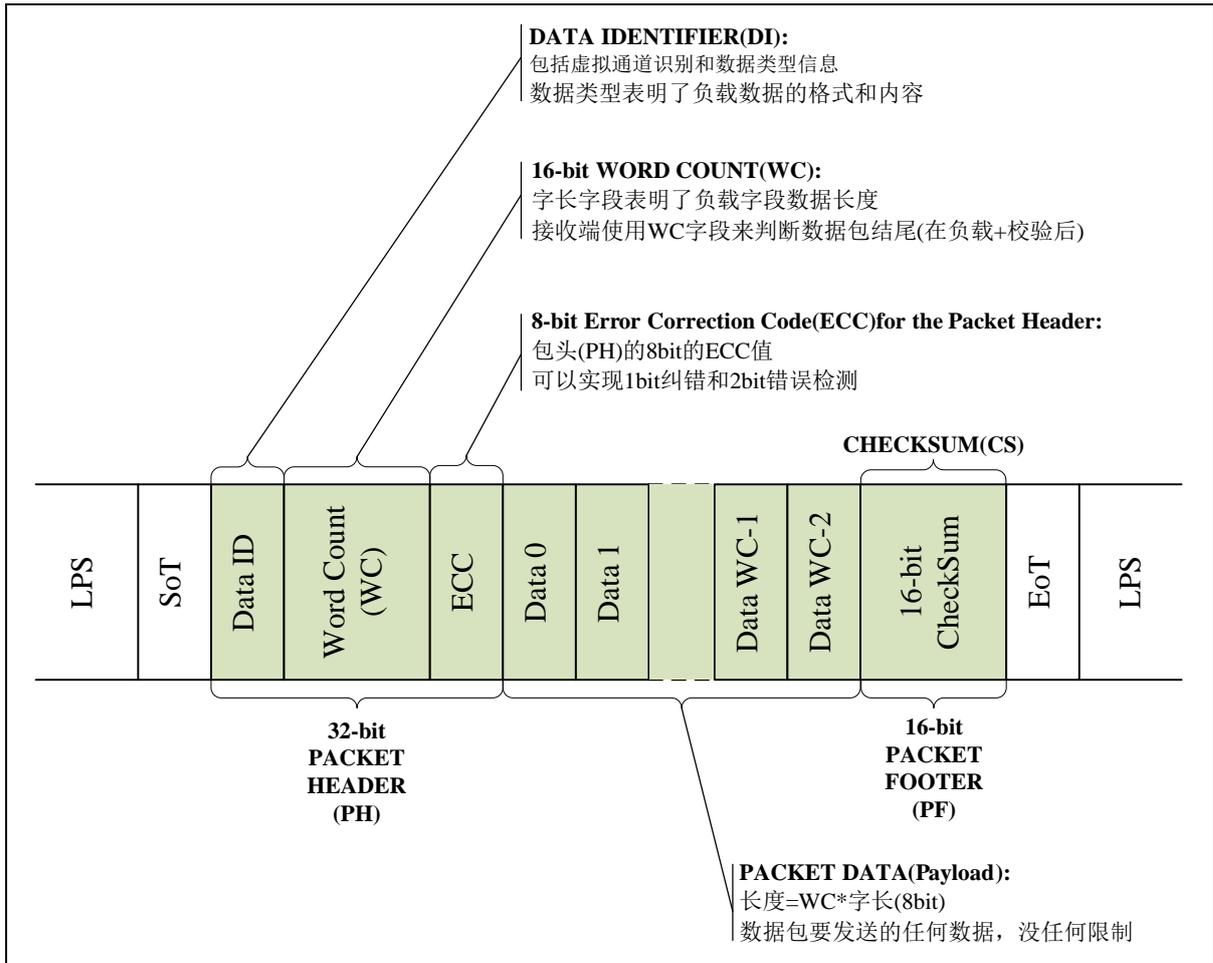


图 24.8.3.1 长数据包结构

图 24.8.3.1 中绿色的部分就是长包结构，长包有 3 部分：32-bit 的 PH(包头)、用于自定义的负载数据、16-bit 的包尾(PF)。PH 有 3 部分：8-bit 的 DI，16-bit 的 WC 以及 8-bit 的 ECC。PF 只有 1 部分：16-bit 的校验和，因此长包数据长度范围是 6~65541 个字节。

24.8.4 短数据包

短包结构如图 24.8.4.1 所示：

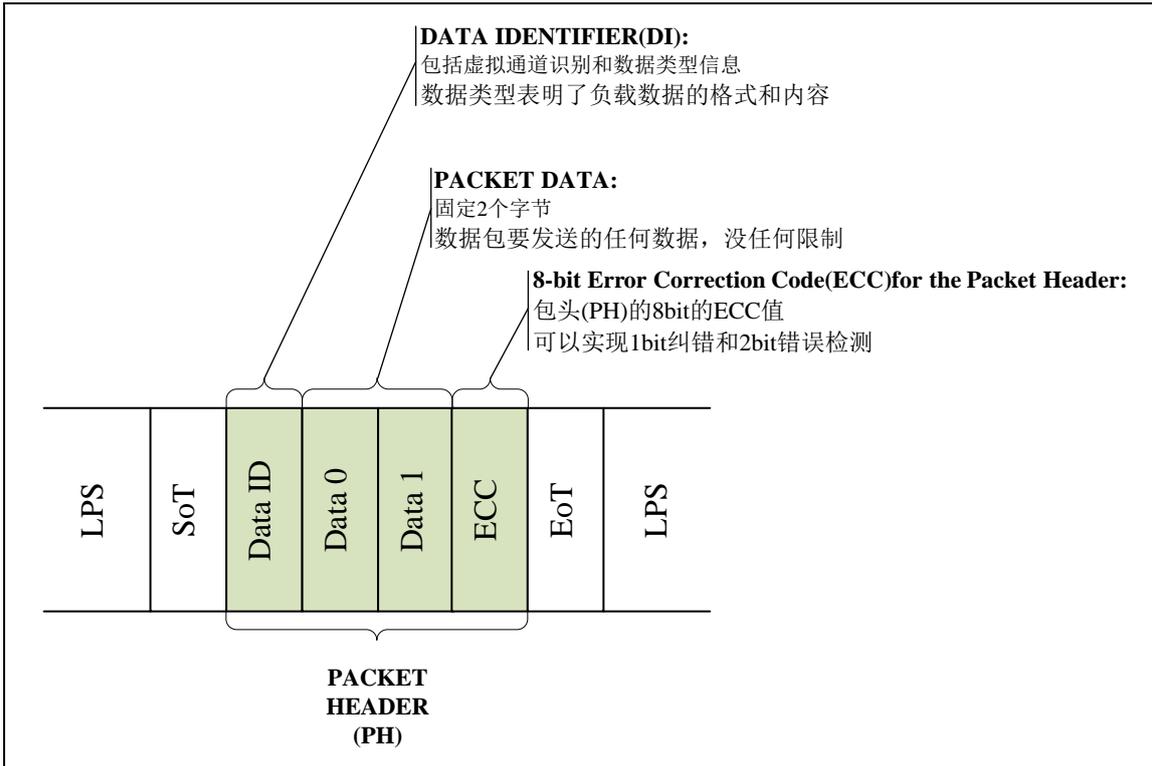


图 24.8.4.1 短数据包结构

图 24.8.4.1 就是短数据包结构，只有一个 PH(包头)，PH 分为 3 部分：和长数据包一样，第 1 个就是 8-bit 的 DI 域；接下来是 2 个字节的数据负载域，也就是用户要实际发送的内容；最后是一个 8-bit 的 ECC 域，可以实现 1bit 纠错，2 比特错误检测。

我们重点来看一下 DI 域，因为长短数据包第一部分就是 DI 域，而且是含义是相同的。DI 域由两部分组成：虚拟通道和负载数据类型，结构如图 24.8.3.2 所示：

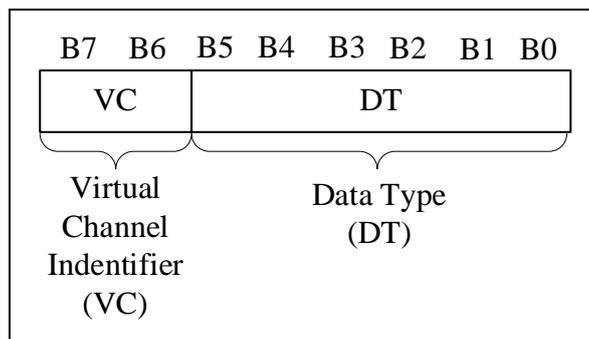


图 24.8.3.2 DI 域接口

其中 bit7:6 这两位指定虚拟通道，在本章节我们不研究这个虚拟通道。bit5:0 这 6 位就是指定后面要发送的负载数据类型，也就是那些负载数据是做啥的。注意，这个数据类型才是我们要学习的重点，后面小节会详细讲解 MIPI DSI 协议所支持的数据类型。

24.9 指令类型

在上一小节讲解长短数据包的时候说过，DI 域包含了后面负载数据的数据类型，MIPI DSI 协议已经定义好了这些类型，这里只介绍最常用的。这里叫说数据类型，有些资料也叫做指令，有两种指令集：Generic 指令和 DSC 指令，这两个的区别在于 Generic 指令是不区分 index 和

parameter 的, 而 DCS 会默认把 data0 作为 index, 然后计算 parameter 数目。关于 DCS 相关的详细请参考《Specification for Display Command Set (DCS).pdf》。

指令分为主机发向外设, 以及外设发向主机两种, 我们要初始化屏幕肯定用的是主机发向外设的, 其指令集合如图 24.9.1 所示:

Data Type, hex	Data Type, binary	Description	Packet Size
0x01	00 0001	Sync Event, V Sync Start	Short
0x11	01 0001	Sync Event, V Sync End	Short
0x21	10 0001	Sync Event, H Sync Start	Short
0x31	11 0001	Sync Event, H Sync End	Short
0x07	00 0111	Compression Mode Command	Short
0x08	00 1000	End of Transmission packet (EoTp)	Short
0x02	00 0010	Color Mode (CM) Off Command	Short
0x12	01 0010	Color Mode (CM) On Command	Short
0x22	10 0010	Shut Down Peripheral Command	Short
0x32	11 0010	Turn On Peripheral Command	Short
0x03	00 0011	Generic Short WRITE, no parameters	Short
0x13	01 0011	Generic Short WRITE, 1 parameter	Short
0x23	10 0011	Generic Short WRITE, 2 parameters	Short
0x04	00 0100	Generic READ, no parameters	Short
0x14	01 0100	Generic READ, 1 parameter	Short
0x24	10 0100	Generic READ, 2 parameters	Short
0x05	00 0101	DCS Short WRITE, no parameters	Short
0x15	01 0101	DCS Short WRITE, 1 parameter	Short
0x06	00 0110	DCS READ, no parameters	Short
0x16	01 0110	Execute Queue	Short
0x37	11 0111	Set Maximum Return Packet Size	Short
0x09	00 1001	Null Packet, no data	Long
0x19	01 1001	Blanking Packet, no data	Long
0x29	10 1001	Generic Long Write	Long
0x39	11 1001	DCS Long Write/write_LUT Command Packet	Long
0x0A	01 1010	Picture Parameter Set	Long
0x0B	00 1011	Compressed Pixel Stream	Long
0x0C	00 1100	Loosely Packed Pixel Stream, 20-bit YCbCr, 4:2:2 Format	Long
0x2C	10 1100	Packed Pixel Stream, 16-bit YCbCr, 4:2:2 Format	Long
0x0D	00 1101	Packed Pixel Stream, 30-bit RGB, 10-10-10 Format	Long
0x1D	01 1101	Packed Pixel Stream, 36-bit RGB, 12-12-12 Format	Long
0x3D	11 1101	Packed Pixel Stream, 12-bit YCbCr, 4:2:0 Format	Long
0x0E	00 1110	Packed Pixel Stream, 16-bit RGB, 5-6-5 Format	Long
0x1E	01 1110	Packed Pixel Stream, 18-bit RGB, 6-6-6 Format	Long
0x2E	10 1110	Loosely Packed Pixel Stream, 18-bit RGB, 6-6-6 Format	Long
0x3E	11 1110	Packed Pixel Stream, 24-bit RGB, 8-8-8 Format	Long
0xX0 and 0xFF, unspecified	XX 0000 XX 1111	DO NOT USE All unspecified codes are reserved	

图 24.9.1 主控向外设发送的命令

图 24.9.1 中有 DCS 指令, 也有 Generic 指令。每个指令是长数据包还是短数据包, 在最后一列都注明了。图中这些指令的详细含义请参考《MIPI DSI Specification_v1-3.pdf》的“8.8 Processor-to-Peripheral Transactions – Detailed Format Description”小节。

我们稍后在具体初始化 MIPI 屏幕的时候就要用到上面这些指令。

24.10 MIPI DSI 时钟计算

根据图 24.6.2.1 的 LCD 结构图可以得到, 在给屏幕传输一帧图形数据的时候要用到的理论像素时钟不是 1280*720, 因为屏幕四周还有一圈“黑边”, 也就是 HBP、HFP、VBP 和 VFP 等, 而且 HSYNC 和 VSYNC 信号也有一定的宽度, 因此屏幕真实的水平和垂直像素数时钟如下:

$$\text{水平: } H\text{-total} = HSYNC + HBP + HACTIVE + HFP$$

$$\text{垂直: } V\text{-total} = VSYNC + VBP + VACTIVE + VFP$$

其中, HSYNC 和 VSYNC 是水平和垂直同步信号宽度, HACTIVE 是屏幕的水平有效像素, VACTIVE 是屏幕的宽度有效像素。HBP、HFP、VBP 和 VFP 这四个参数前面已经说过了, 这些参数都能在屏幕手册里面找到, 比如正点原子两款 5.5 寸 MIPI 屏幕参数如表 24.10.1 所示:

5.5 寸 MIPI 720*1280 屏幕		
时序参数	数值	说明
hactive	720	水平分辨率
vactive	1280	垂直分辨率
hfp	48	水平前肩
hbp	52	水平后肩
hsync-len	8	水平同步信号宽度
vfp	16	垂直前肩
vbp	15	垂直后肩
vsync-len	6	垂直同步信号宽度
5.5 寸 MIPI 1080*1920 屏幕		
时序参数	数值	说明
hactive	1080	水平分辨率
vactive	1920	垂直分辨率
hfp	45	水平前肩
hbp	5	水平后肩
hsync-len	45	水平同步信号宽度
vfp	9	垂直前肩
vbp	3	垂直后肩
vsync-len	4	垂直同步信号宽度

表 24.10.1 正点原子 MIPI 屏幕时序参数

1 秒钟的像素数量就是:

$$Pixel - total = H - total \times V - total \times fps$$

其中 fps 就是屏幕帧率, 一般是 60 帧, 也就是 1 秒钟刷新 60 张图像, Pixel-total 就是 1 秒钟要传输的总的像素点。

如果屏幕采用 RGB888 格式, 那么 1 个像素就是 24bit, 如果是 RGB565 那就是 16bit, 这个叫色深, 我们一般都使用 RGB888。那么总的 bit 时钟就是:

$$Bit\text{-total} = Pixel\text{-total} \times \text{色深(比如)24 或 16}$$

得到的 Bit-total 是总的 bit 时钟数, 但是 MIPI DSI 可以配置多条数据 lane, 一般是 2 或 4lane, 也就是有 2 个或者 4 个通道, 所以每个通道的 bit 时钟就是:

$$\text{Bit-clk} = \frac{\text{Bit-total}}{\text{Lane number}}$$

计算出来的 Bit-clk 就是 MIPI DSI 的时钟, 但是由于 MIPI DSI 是双边沿采集, 所以最终的 DSI CLK 时钟还要除以 2:

$$\text{Dsi-clk} = \frac{\text{Bit-clk}}{2}$$

总结一下 DSI CLK 时钟的计算公式如下:

$$\text{Dsi-clk} = \frac{(\text{HSYNC} + \text{HBP} + \text{HACTIVE} + \text{HFP}) \times (\text{VSYNC} + \text{VBP} + \text{VACTIVE} + \text{VFP}) \times \text{fps} \times \text{色深}}{\text{lane number}} \times \frac{1}{2}$$

我们以正点原子 720*1280 这款 5.5 寸 MIPI 屏幕为例, 采用 RGB888 格式, 帧率为 60fps, 使用 4lane 传输数据, 那么此屏幕的 DSI 时钟为:

$$\begin{aligned} \text{Dsi-clk} &= (8+52+720+48) \times (6+15+1280+16) \times 60 \times 24/4/2 \\ &= 828 \times 1317 \times 60 \times 24/4/2 \\ &= 196,285,680\text{Hz} \\ &\approx 197\text{MHz} \end{aligned}$$

197M 就是我们用示波器测出来的频率, 注意这个 197M 只是理论值, 实际值要高! 因为要考虑到开销, 后面会讲。

我们一般说的 MIPI DSI 时钟要在这个时间测量到的频率上乘以 2, 因为双边沿采集嘛, 所以 MIPI DSI 速率就是 $196285680 \times 2 = 392571360\text{Hz} \approx 393\text{M}$ 。

注意重点:

我们最终需要在设备树里面设置 MIPI DSI 速率, 但是不能直接设置前面计算出来的理论 MIPI DSI 速率, 我们在前面学习长短包的时候就知道, 实际的 MIPI DSI 通信中还有其他的开销。加入直接将 MIPI DSI 的速率设置成 393M, 那么实际的屏幕帧率肯定到不了 60fps。但是我们也没必要研究具体用了多少开销, 实际精准的速率是多少, 这样太耗费时间了, 难度也很大。一般都是在理论速率上加上一些余量, 网上有些资料说对于瑞芯微的平台, 实际设置的 MIPI DSI 速度是理论的 1.2 倍即可, 比如 $393 \times 1.2 = 468\text{M}$, 那么实际设置的 MIPI DSI 速率大于 468M 即可。

24.11 RK3568 MIPI DSI 介绍

RK3568 有一个 MIPI DSI 接口, 所以也有一个 MIPI DSI 主控外设, 用于完成 MIPI DSI 屏幕的驱动。此 MIPI DSI HOST 内核符合 MIPI 协议, MIPI DSI HOST 用于连接内核和 D-PHY, RK3568 的 MIPI DSI HOST 接口支持 1~4Lane。

RK3568 的 MIPI DSI HOST 控制器支持的特性如下:

- 兼容 MIPI 联盟标准。
- 支持 DPI 接口颜色映射, 支持 16/18/24bit 色深。
- 所有 DPI 接口信号极性可编程
- 最高支持 4Lane 的 D-PHY 数据 Lane
- Data0 支持双线通信和 Escape 模式
- 可以传输所有的 Generic 命令
- 支持 EOTP 包
-

RK3568 的 MIPI DSI HOST 控制器框图如图 24.11.1 所示:

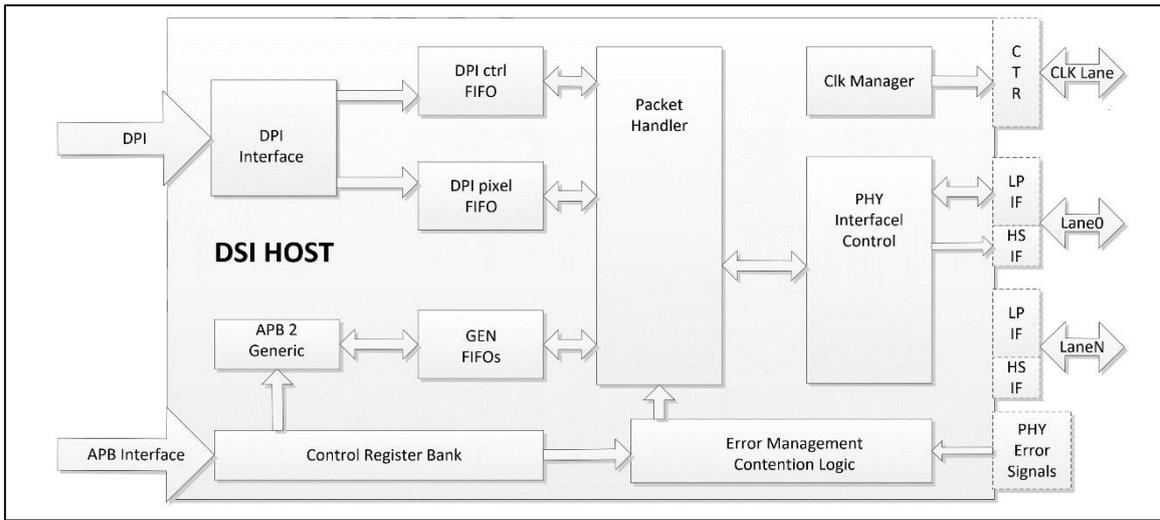


图 24.11.1 RK3568 MIPI DSI HOST 框图

RK3568 的 D-PHY 特性如下:

- V1.2 版本的 MIPI D-PHY。
- 集成 PPI 接口, 支持 DSI。
- 每条 Lane 最高 2.5Gbps 传输速率。
- 最多支持 4Lane, 总共 10.0Gbps 传输速率。
- 支持 MIPI HS 和 LP 模式。
- 支持 Skew 校准。
- LP 模式下 10Mbps 传输速率。
-

关于 RK3568 的 MIPI DSI 外设就介绍到这里, 具体的内容可以自行查阅 RK3568 参考手册。

24.12 硬件原理图分析

我们先分析一下正点原子 ATK-DLRK3568 开发板的 MIPI 屏幕硬件原理图, 底板上 MIPI DSI 屏幕原理如图 24.12.1 所示:

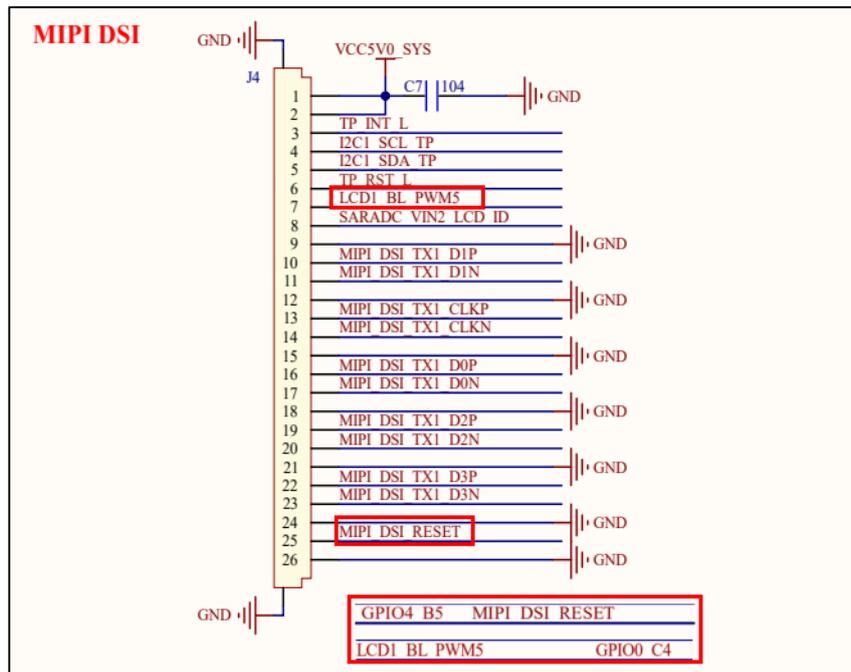


图 24.12.1 MIPI DSI 屏幕接口原理图

从图 24.12.1 可以看出，正点原子 ATK-DLRK3568 开发板的 MIPI 屏幕接口采用 4Lane，其中 MIPI_RESET 是屏幕的复位引脚，连接到 RK3568 的 GPIO4_B5 这个引脚上。LCD_1BL_PWM5 是屏幕背光 PWM 引脚，连接到 RK3568 的 GPIO0_C4 引脚上。

24.13 MIPI 屏幕驱动调试思路

本小节是笔者根据自己调试 RK3568 MIPI 屏幕的时候总结出来的思路，希望对大家有借鉴作用。

1、必须和 SOC 以及屏幕原厂合作

首先非常重要的一点就是，一定要和 SOC 和屏幕原厂沟通交流，因为他们对自己的产品最熟悉，很多时候都能够给出一针见血的意见。这个时候，很多小公司或者个人开发者就会说他们根本没法和 SOC 或屏幕原厂沟通，因为量少原厂根本就不理他们。确实，这个问题是存在的，这个时候 MIPI 屏幕驱动调试就比较麻烦了，只能祈祷你所使用的屏幕以及芯片自带的驱动没有问题。

我们公司累计在 RV1126、RK3568、STM32MP157、全志 D1 等多种平台上调试过多款屏幕，在调试过程中都是和 SOC 以及屏幕原厂直接进行沟通。当然了，只要一款调通了，其他的调试起来都是大同小异的。我们调试最花时间的就是第一批定制屏幕，由于是定制屏幕，所以可能屏幕本身也有一些问题，调试起来磕磕绊绊，这个时候屏幕原厂的协助就尤为重要！如果你所使用的屏幕是已经大批量供货的，那么调试起来会轻松很多。

另外我们基本没有和 SOC 原厂沟通过，比如瑞芯微，我们在调试 RV1126、RK3568 这些平台的时候都是自己找资料搞定的。因为 SOC 原厂提供的 SDK 里面对于 MIPI DSI 主控端驱动已经是很完善的了，是不需要我们修改什么的。我们要做的就是搞明白不同 SOC 其设备树下如何添加 MIPI 屏幕配置参数即可。

2、必备逻辑分析仪

最好准备一个支持 MIPI DSI LP 的逻辑分析仪，因为 MIPI 屏幕驱动的一个重点就是主控

向屏幕发送初始化序列。前面在讲 MIPI 协议的时候就说了，会通过 Data0 这对 Lane 来完成这个初始化操作，而且是 LP 模式下。正点原子逻辑分析仪支持 MIPI DSI LP 协议分析，这个是调试 MIPI 屏幕的杀手锏，可以直接抓取实际的信号波形，分析初始化序列发送是否成功。如果屏幕初始化序列发送没问题，那么 MIPI 屏幕驱动就完成大半了。

3、三大调试内容

调试 MIPI 屏幕主要有三部分内容：

1)、屏幕背光调试，这个是首先要搞定的，背光不亮，屏幕也就什么都看不到。这个比较简单，属于 PWM 相关知识，后面也会讲解如何调试背光。

2)、向屏幕发送初始化序列，前面已经说了这点，需要用到逻辑分析仪。

3)、调试屏幕的 DPI 参数，最后需要调试 MIPI 屏幕的 DPI 参数，也就是 HBP、HFP、VBP、VFP 等这些参数。

我们后面编写驱动的时候也是按照这三大内容逐步调试的。

24.14 实验程序编写

24.14.1 背光驱动

1、背光 PWM 节点设置

屏幕背光使用 PWM 来控制，通过 PWM 波形来调节屏幕亮度。关于 PWM 已经在《第二十三章 Linux PWM 驱动实验》进行了详细的讲解。

正点原子的 MIPI DSI 屏幕接口背光控制 IO 连接到了 RK3568 的 GPIO0_C4 引脚上，我们需要将 GPIO0_C4 复用为 PWM5，然后通过此 PWM 信号来控制屏幕背光的亮度，接着我们来看一下如何在设备树中添加背光节点信息。

首先是 GPIO0_C4 这个 pinctrl 的配置，在 rk3568-pinctrl.dtsi 中找到如下内容：

示例代码 24.14.1.1 GPIO0_C4 的 pinctrl 配置

```

1  pwm5 {
2      /omit-if-no-ref/
3      pwm5_pins: pwm5-pins {
4          rockchip,pins =
5              /* pwm5 */
6              <0 RK_PC4 1 &pcfg_pull_none>;
7      };
8  };
    
```

示例代码 24.14.1.1 第 3 行的“pwm5_pins”就是引脚 pinctrl 节点名字，在第 6 行将 GPIO0_C4 引脚设置为 PWM5 功能，默认禁用引脚的上下拉电阻，即不启用上拉和下拉。

继续在 rk3568.dtsi 文件中向 pwm5 追加内容，如下所示：

示例代码 24.14.1.2 向 pwm0 节点追加内容

```

1  pwm5: pwm@fe6e0010 {
2      compatible = "rockchip,rk3568-pwm", "rockchip,rk3328-pwm";
3      reg = <0x0 0xfe6e0010 0x0 0x10>;
4      #pwm-cells = <3>;
5      pinctrl-names = "active";
6      pinctrl-0 = <&pwm5_pins>;
    
```

```

7     clocks = <&cru CLK_PWM1>, <&cru PCLK_PWM1>;
8     clock-names = "pwm", "pclk";
9     status = "disabled";
10  }
    
```

第 6 行, 设置使用示例代码 24.14.1.1 中的 pwm5_pins 这个引脚配置节点。

2、backlight 节点设置

到这里, PWM 和相关的 IO 已经准备好了, 但是 Linux 系统怎么知道 pwm5 就是控制屏幕背光的呢? 因此我们还需要一个节点来将屏幕背光和 pwm0_m0 连接起来。这个节点就是 backlight, backlight 节点描述可以参考 [Documentation/devicetree/bindings/leds/backlight/pwm-backlight.txt](#) 这个文档, 此文档详细讲解了 backlight 节点该如何去创建, 这里大概总结一下:

①、节点名称要为“backlight”。

②、节点的 compatible 属性值为“pwm-backlight”, 因此可以通过在 Linux 内核中搜索“pwm-backlight”来查找 PWM 背光控制驱动程序, 这个驱动程序文件为 drivers/video/backlight/pwm_bl.c, 感兴趣的可以去看一下这个驱动程序。

③、pwms 属性用于描述背光所使用的 PWM 的通道以及 PWM 频率, 比如本章我们要使用的 pwm5, pwm 频率设置为 40KHz。那么 pwms 属性就可以设置为:

```
pwms = <&pwm5 0 25000 0>
```

一共有 4 个参数, 第 1 个“pwm5”表示使用 PWM5; 第 2 个“0”就是使用通道 0, 也就是 PWM5; 第 3 个 25000 是 PWM 的周期, 单位是 ns, 也就是 25000ns, 换算成频率就是 40KHz; 最后一个 0 是极性设置, 可以设置为 PWM_POLARITY_NORMAL(对应的值为 0)或者 PWM_POLARITY_INVERTED(对应的值为 1), 第一个是正常极性, 第二个极性翻转。

④、brightness-levels 属性描述亮度级别, 范围为 0~255, 0 表示 PWM 占空比为 0%, 也就是亮度最低, 255 表示 100%占空比, 也就是亮度最高。至于设置几级亮度, 大家可以自行填写此属性。

④、default-brightness-level 属性为默认亮度级别。

根据上述 5 点设置 backlight 节点, 我们在根节点创建一个 backlight 节点, 在 rk3568-evb.dtsi 文件中新建内容如下 (注因有多种屏幕, 所以我们的 MIPI 屏幕使用的背光节点命名为 backlight1):

示例代码 24.14.1.3 backlight 节点

```

109 backlight1: backlight1 {
110     compatible = "pwm-backlight";
111     pwms = <&pwm5 0 25000 0>;
112     brightness-levels = <
113         0 20 20 21 21 22 22 23
114         23 24 24 25 25 26 26 27
115         27 28 28 29 29 30 30 31
116         31 32 32 33 33 34 34 35
117         35 36 36 37 37 38 38 39
118         40 41 42 43 44 45 46 47
119         48 49 50 51 52 53 54 55
120         56 57 58 59 60 61 62 63
121         64 65 66 67 68 69 70 71
122         72 73 74 75 76 77 78 79
    
```

```

123         80 81 82 83 84 85 86 87
124         88 89 90 91 92 93 94 95
125         96 97 98 99 100 101 102 103
126        104 105 106 107 108 109 110 111
127        112 113 114 115 116 117 118 119
128        120 121 122 123 124 125 126 127
129        128 129 130 131 132 133 134 135
130        136 137 138 139 140 141 142 143
131        144 145 146 147 148 149 150 151
132        152 153 154 155 156 157 158 159
133        160 161 162 163 164 165 166 167
134        168 169 170 171 172 173 174 175
135        176 177 178 179 180 181 182 183
136        184 185 186 187 188 189 190 191
137        192 193 194 195 196 197 198 199
138        200 201 202 203 204 205 206 207
139        208 209 210 211 212 213 214 215
140        216 217 218 219 220 221 222 223
141        224 225 226 227 228 229 230 231
142        232 233 234 235 236 237 238 239
143        240 241 242 243 244 245 246 247
144        248 249 250 251 252 253 254 255
145        >;
146        default-brightness-level = <255>;
147    };
    
```

第 111 行, 设置背光使用 `pwm5`, PWM 频率为 40KHz。

第 112~145 行, 设置 256 级背光(0~255)。

第 146 行, 设置默认背光等级为 255(最亮)。

至此, 屏幕背光就设置好了。

注意! 屏幕背光 PWM 设置好以后屏幕可能不会亮, 但是你测量 PWM 信号又是正常的, 不要急, 有可能是因为 MIPI 屏幕 IC 还没初始化, 导致背光部分有问题, 背光电压不正常。大家可以测量一下背光电压是否正常, 如果背光电压不正常那么基本就是因为 MIPI 屏幕还没初始化。笔者及同事在调试 MIPI 屏幕的时候就遇到过这种问题, 因为屏幕 IC 还没初始化, 导致背光电压不正常, 最后联系屏幕厂商得到的回复就是要正确初始化屏幕 IC, 也就是下面要做的向屏幕发送初始化序列。

24.14.2 屏幕初始化序列发送时序参数设置

接下来就需要向 MIPI 屏幕 IC 发送初始化序列, 也就是在 LP 模式下, 进入 `Escape+LPDT` 指令来发送初始化序列, 这个过程可以使用正点原子逻辑分析抓取波形, 并且进行协议分析, 后面会有一节专门讲解时序分析。在 MIPI 屏幕调试过程中, 使用逻辑分析仪采集并分析 MIPI DSILP 协议非常重要, 因为只有这样才能清楚的知道主控实际上向屏幕发送的数据是否正确!

1、设备树下 DSI 节点编写

MIPI 屏幕的初始化序列发送相关操作都是在设备树里面完成的,也就是我们要设置好 DSI 接口的设备树节点,以及添加要发送的初始化序列。所以我们要先学习一下相关的设备树内容,瑞芯微芯片的 MIPI DSI 设备树绑定信息文档为: [Documentation/devicetree/bindings/display/rockchip/dw_mipi_dsi_rockchip.txt](#),我们简单总结一下瑞芯微芯片的 DSI 节点信息。

①、必须属性:

#address-cells: 必须为 1。

#size-cells: 必须为 0。

compatible: 可以是下面这些中的某一个:

"rockchip,px30-mipi-dsi", "snps,dw-mipi-dsi".

"rockchip,rk1808-mipi-dsi", "snps,dw-mipi-dsi".

"rockchip,rk3128-mipi-dsi", "snps,dw-mipi-dsi".

"rockchip,rk3288-mipi-dsi", "snps,dw-mipi-dsi".

"rockchip,rk3368-mipi-dsi", "snps,dw-mipi-dsi".

"rockchip,rk3399-mipi-dsi", "snps,dw-mipi-dsi".

"rockchip,rk3568-mipi-dsi", "snps,dw-mipi-dsi".

"rockchip,rv1126-mipi-dsi", "snps,dw-mipi-dsi".

对于 RK3568 而言,用的“rockchip,rk3568-mipi-dsi”这个。

reg: RK3568 的 DSI 控制器物理寄存器基地址,为 0xfe070000,这个可以在 RK3568 的数据手册上找到。

interrupts: DSI 的中断控制器。

power-domains: 电源域句柄。

resets: 复位句柄。

reset-names: 复位名字,必须是“apb”。

clocks: 时钟来源。

clock-names: 时钟名字。

rockchip,grf: SOC 的 grf 寄存器。

ports: 这是一个名为“ports”的子节点,

②、可选属性:

phys: DSI 所使用的 PHY,对于 RK3568 而言是 DPHY。

phy-names: 对于 RK3568 而言是“mipi_dphy”。

rockchip,lane-rate: MIPI 总线速率,用示波器测量的时候,实际波形频率是此速率的一半。
注意在 RK 3 5 6 8 此项已经不使用些属性,会根据配置的参数自动计算。

2、DSI 的 panel 子节点编写

上面是 RK3568 的 DSI 主控节点,在 DSI 节点下还有一个非常重要的子节点,名字叫做“panel”,panel 子节点用来描述屏幕相关信息,其中就包括屏幕的初始化序列以及屏幕的时序参数,关于 panel 节点的设备树绑定信息文档为: [Documentation/devicetree/bindings/display/panel/simple-panel.txt](#)。简单总结一下 panel 节点信息。

①、必须属性:

compatible: 驱动兼容属性,可以是下面这些中的某一个:

“simple-panel”: 通用的 panel

“simple-panel-dsi”: 通用的 DSI panel

“vendor.panel”: 明确厂商的 panel

RK3568 的 DSI 屏幕用的 compatible 属性值就是 “simple-panel-dsi”。

power-supply: panel 的电压要求。

②、可选属性:

enable-gpios: panel 使能引脚。

reset-gpios: panel 复位引脚。

backlight: panel 要使用的背光句柄, 也就是示例代码 24.14.1.3 中的 backlight 节点。

prepare-delay-ms: panel 准备就绪并开始接收视频数据的时间, 单位为 ms。

enable-delay-ms: panel 开始接收视频数据到显示出第一帧画面所用的时间, 单位为 ms。

disable-delay-ms: panel 关闭显示花费的时间, 单位为 ms。

unprepare-delay-ms: panel 完全断电所花费的时间, 单位为 ms。

reset-delay-ms: panel 完全复位所花费的时间, 单位为 ms。

init-delay-ms: panel 复位到发送初始化序列之间的时间, 单位为 ms。

width-mm: panel 宽度, 也就是屏幕的物理宽度, 单位 mm。

height-mm: panel 高度, 也就是屏幕的物理高度, 单位 mm。

bpc: 色深。

bus-format: 数据线上的像素格式。

dsi,lanes: DSI 接口的 lane 数量。

dsi,format: video 模式下像素格式。

dsi,flags: 可选的一些标记信息。

panel-init-sequence 和 panel-exit-sequence: 这两个很重要, 尤其是 panel-init-sequence, 也就是屏幕的初始化序列, 在初始化屏幕的时候 DSI 主控在 LP 模式下向屏幕发送的初始化序列就保存在 panel-init-sequence 中。但是不是简单的将要发送的序列一次排开, 一股脑填写到 panel-init-sequence 里面, 而是按照规格要求填写, 初始化序列字节流要求如下:

Byte 0: DCS 数据类型。

Byte 1: 等到多少 ms 发送 DCS 数据包。

Byte 2: 负载数据长度, 也就是真正要发送给屏幕 IC 的数据。

Byte 3 及以后: 负载数据, 也就是要发送个屏幕 IC 的初始化序列。

初始化序列可能要做很多不同的操作, 用到不同的 DCS 指令, 所以会有很多行, 一行代表一种操作, 每行都按照上面的字节流要求填写。

最后还需要设置 MIPI 屏幕的 DPI 时序参数, 在 panel 节点下创建一个子节点 “display-timings”, 用来设置 MIPI 的 DPI 时序参数。DPI 时序参数设置比较简单, 就是以前那种 RGB 屏幕。

24.14.3 DSI 设备树节点编写

上一小节已经详细讲解了如何编写 DSI 对应的节点, 包括节点里面的属性都是什么意思, 本节我们就根据正点原子 ATK-DLRK3568 开发板和配套屏幕看一下实际的节点内容。在 rk3568-evb.dtsi 文件中找到如下内容:

示例代码 24.14.3.1 dsi1 节点追加内容

```
740     &dsi1 {
741         status = "disabled";
```

```

742     //rockchip, lane-rate = <1000>;
743     dsil_panel: panel@0 {
744         status = "okay";
745         compatible = "simple-panel-dsi";
746         reg = <0>;
747         backlight = <&backlight1>;
748         reset-delay-ms = <60>;
749         enable-delay-ms = <60>;
750         prepare-delay-ms = <60>;
751         unprepare-delay-ms = <60>;
752         disable-delay-ms = <60>;
753         dsi, flags = <(MIPI_DSI_MODE_VIDEO |
754             MIPI_DSI_MODE_VIDEO_BURST |
755             MIPI_DSI_MODE_LPM | MIPI_DSI_MODE_EOT_PACKET)>;
756         dsi, format = <MIPI_DSI_FMT_RGB888>;
757         dsi, lanes = <4>;
758         panel-init-sequence = [
759             23 00 02 FE 21
760             23 00 02 04 00
761             23 00 02 00 64
762             ... ..
763             23 00 02 20 71
764             23 00 02 50 8F
765             23 00 02 51 8F
766             23 00 02 FE 00
767             23 00 02 35 00
768             05 78 01 11
769             05 1E 01 29
770         ];
771
772         panel-exit-sequence = [
773             05 00 01 28
774             05 00 01 10
775         ];
776
777         disp_timings1: display-timings {
778             native-mode = <&dsil_timing0>;
779             dsil_timing0: timing0 {
780                 clock-frequency = <132000000>;
781                 hactive = <1080>;
782                 vactive = <1920>;
783                 hfront-porch = <15>;
784                 hsync-len = <2>;
    
```

```

1034         hback-porch = <30>;
1035         vfront-porch = <15>;
1036         vsync-len = <2>;
1037         vback-porch = <15>;
1038         hsync-active = <0>;
1039         vsync-active = <0>;
1040         de-active = <0>;
1041         pixelclk-active = <1>;
1042     };
1043 };
1044
1045     ports {
1046         #address-cells = <1>;
1047         #size-cells = <0>;
1048
1049         port@0 {
1050             reg = <0>;
1051             panel_in_dsi1: endpoint {
1052                 remote-endpoint = <&dsil_out_panel>;
1053             };
1054         };
1055     };
1056 };
1057
1058     ports {
1059         #address-cells = <1>;
1060         #size-cells = <0>;
1061
1062         port@1 {
1063             reg = <1>;
1064             dsil_out_panel: endpoint {
1065                 remote-endpoint = <&panel_in_dsi1>;
1066             };
1067         };
1068     };
1069
1070 };
    
```

第 740 行, MIPI 使用的是 DSI1。

第 741 行, 这里的 `status = "disabled"`; 表示禁用, 但是我们在 `rk3568-atk-evb1-mipi-dsi-720p.dts/rk3568-atk-evb1-mipi-dsi-1080p.dts` 中设置 `dsi1` 启用, 并重写了 `dsi1` 里的参数。所以我们在这里看到的参数基本都会被重写。

第 742 行, 设置 MIPI 速率, 这里在 RK3568 上已经不启用了, 因为会根据屏幕配置的时钟参数自动计算, 大家可以看看前面的计算公式。

第 757~1043 行就是 panel 子节点, 初始化 MIPI 屏幕的具体参数就在此节点里面。我们重点看一下 panel 子节点下的各个属性。

第 747 行, 设置屏幕背光使用示例代码 24.14.1.3 中的 backlight 节点。

第 748~752 行, 一些时间相关的设置, 具体含义看上一小节中相关设备树节点属性讲解。

第 753~754 行, 设置 MIPI 屏幕一些模式:

MIPI_DSI_MODE_VIDEO: 主控采用 video 模式。

MIPI_DSI_MODE_VIDEO_BURST: 使用 video 的 Burst 模式。

MIPI_DSI_MODE_LPM: 支持 LP 模式, 也就是在 LP 模式下传输初始化序列。

MIPI_DSI_MODE_EOT_PACKET: 禁止 HS 模式下的 EoT 包。

第 755 行, 屏幕采用 RGB888 格式。

第 756 行, 使用 4lanes。

前面说了, 在这个 rk3568-evb.dtsi 配置的 panel-init-sequence 和 disp_timings1 都会被重写, 因为正点原子有几个分辨率的屏幕, 分别是 5.5 英寸 720x1280MIPI 屏, 5.5 英寸 1080x1920MIPI 屏幕, 10.1 英寸 800x1280MIPI 屏幕, 所以写在一个 dts 里肯定是不实际的。rk3568-evb.dtsi 里的 dsi1 配置就是模板, 现在我们要重写这里面的参数, 以 720P 的 MIPI 屏幕为例。打开 rk3568-lcds.dtsi, 找到以下代码。

示例代码 24.14.3.2 dsi1_panel 节点内容

```

161 &dsi1_panel {
162     status = "okay";
163     panel-init-sequence = [ //720p mipi 屏参数
164         39 00 04 B9 FF 83 94
165         39 00 07 BA 63 03 68 6B B2 C0
166         //15 00 02 36 01(倒向显示)
167         //15 00 02 36 02(正向显示)
168         15 00 02 36 01
169         39 00 0B B1 48 12 72 09 32 54 71 71 57 47
170         39 00 07 B2 00 80 64 0C 0D 2F
171         39 00 16 B4 73 74 73 74 73 74 01 0C 86 75 00 3F 73 74
73 74 73 74 01 0C 86
172         39 00 03 B6 6E 6E
173         39 00 22 D3 00 00 07 07 40 07 0C 00 08 10 08 00 08 54
15 0A 05 0A 02 15 06 05 06 47 44 0A 0A 4B 10 07 07 0C 40
174         39 00 2D D5 1C 1C 1D 1D 00 01 02 03 04 05 06 07 08 09
0A 0B 24 25 18 18 26 27 18 18 18 18 18 18 18 18 18 18 18 18 18 18
20 21 18 18 18 18
175         39 00 2D D6 1C 1C 1D 1D 07 06 05 04 03 02 01 00 0B 0A
09 08 21 20 18 18 27 26 18 18 18 18 18 18 18 18 18 18 18 18 18 18
25 24 18 18 18 18
176         39 00 3B E0 00 0A 15 1B 1E 21 24 22 47 56 65 66 6E 82
88 8B 9A 9D 98 A8 B9 5D 5C 61 66 6A 6F 7F 7F 00 0A 15 1B 1E 21 24 22 47
56 65 65 6E 81 87 8B 98 9D 99 A8 BA 5D 5D 62 67 6B 72 7F 7F
177         39 00 03 C0 1F 31
178         15 00 02 CC 03
    
```

```

179         15 00 02 D4 02
180         15 00 02 BD 02
181         39 00 0D D8 FF FF
182         15 00 02 BD 00
183         15 00 02 BD 01
184         15 00 02 B1 00
185         15 00 02 BD 00
186         39 00 08 BF 40 81 50 00 1A FC 01
187         15 00 02 C6 ED
188         05 64 01 11
189         05 78 01 29
190     };
191 };
    
```

第 163~189 行，非常重要！前面说了 panel-init-sequence 描述了要发送给屏幕的初始化序列，其中 164~189 行就是要发送的初始化序列，一共有 24 条。每条初始化序列都要按照上一小节的要求填写字节流，比如第 164 行是“39 00 04 B9 FF 83 94”，其组成结构如图 24.14.3.1 所示：

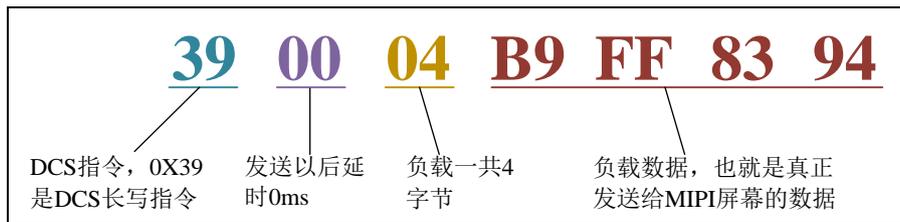


图 24.14.3.1 初始化序列结构图

从图 24.14.3.1 可以看出，真正发送给屏幕初始化序列是“B9 FF 83 94”，这个也就是屏幕厂商给到你的初始化序列。其中第一个 0X39 就是图 24.9.1 中的指令，DCS 长写指令，是个长数据包。后续我们会用逻辑分析仪抓取波形，会发现这个包就是长数据包格式。

我们再来看一下第 188 行这条初始化序列，为“05 64 01 11”，同样使用上面的分析方法，0X05 对应 DCS 短写指令，是个短数据包格式。0X64 表示这条初始化序列发送以后延时 100ms(0X64)再发送下一条；0X01 说明负载只有 1 个字节；0X11 就是最终要发送出去的负载值。

其他初始化序列分析方法相同，大家自行分析。

同时我们也来看看重写的 720P 的时序参数，在 rk3568-atk-evb1-ddr4-v10.dtsi 找到以下代码。

第 343~357 行，display-timings1 就是 MIPI 屏幕的 DPI 时序参数，也就是 HFP、HBP 等等这些时序信息，根据自己所使用的屏幕实际参数填写。

```

示例代码 24.14.3.3 dsil_timing1 节点内容
343     dsil_timing1: timing1 {
344         clock-frequency = <65000000>;
345         hactive = <720>;
346         vactive = <1280>;
347         hfront-porch = <48>;
    
```

```

348         hsync-len = <8>;
349         hback-porch = <52>;
350         vfront-porch = <16>;
351         vsync-len = <6>;
352         vback-porch = <15>;
353         hsync-active = <0>;
354         vsync-active = <0>;
355         de-active = <0>;
356         pixelclk-active = <0>;
357     };
    
```

24.15 运行测试

设备树修改完成以后，编译内核，然后将新的内核烧写到开发板并启动测试。如果设备树修改没有问题，那么大家就能看到 MIPI 屏幕点亮，会显示如图 24.15.1 所示界面：



图 24.15.1 MIPI 屏幕启动显示

图 24.15.1 是在正点原子 ATK-DLRK3568 开发板测试结果，会显示 logo 信息，如果启动了 UI 桌面的话就会显示具体的 UI 界面信息(出厂系统默认是 QT 做的一个界面)。

显示测试很简单，只要看屏幕能不能显示就 OK 了，从学习的角度来讲，我们需要更加深入的测试，比如背光 PWM 如何调节测试，用逻辑分析仪抓取 MIPI DSI LP 协议分析过程等。接下来我们就来详细介绍这些改如何测试。

24.15.1 LCD 背光调节

背光调节不用像上一章 PWM 实验那样，直接操作 pwmchipX(X=0~N)目录里面的文件，但是我们还是需要看一下。因为本章我们开启了 PWM0 这路 PWM，上一章开启了 PWM15 这路 PWM，同上一小节 PWM 实验一样，进入/sys/class/pwm 目录下，如下图所示：

```

root@ATK-DLRK356X:/sys/class/pwm# ls
pwmchip0  pwmchip1  pwmchip2  pwmchip3
root@ATK-DLRK356X:/sys/class/pwm#
    
```

图 24.15.1.1 pwm 文件

从图 24.15.1.1 可以看出此时有 pwmchip0~pwmchip3，上一章 PWM 实验中 pwmchip3 对应

PWM15。同理也是用 ls -l 查看，如图 24.15.1.2 所示：

```
root@ATK-DLRK356X:/sys/class/pwm# ls -l
total 0
lrwxrwxrwx 1 root root 0 Aug 4 2017 pwmchip0 -> ../../devices/platform/fdd70020.pwm/pwm/pwmchip0
lrwxrwxrwx 1 root root 0 Aug 4 2017 pwmchip1 -> ../../devices/platform/fe6e0000.pwm/pwm/pwmchip1
lrwxrwxrwx 1 root root 0 Aug 4 2017 pwmchip2 -> ../../devices/platform/fe6e0010.pwm/pwm/pwmchip2
lrwxrwxrwx 1 root root 0 Aug 4 2017 pwmchip3 -> ../../devices/platform/fe700030.pwm/pwm/pwmchip3
root@ATK-DLRK356X:/sys/class/pwm#
```



图 24.15.1.2 pwmchip2 路径

从图 24.15.1.2 可以看出，此时 pwmchip2 对应的定时器寄存器首地址为 0XFE6E0010，这个正是 PWM5 的寄存器首地址，在手册可以查到！

大家在开启多路 PWM 以后，一定要使用这个的方法来确定 PWM 对应的 pwmchip 文件！！

第 24.14.1 小节已经讲过了，背光设备树节点设置了 256 个等级的背光调节，可以设置为 0~255，我们可以通过设置背光等级来实现 LCD 背光亮度的调节，为什么是 backlight1 呢？RK3568 有多路显示，前面设备树配置的就是 backlight1，进入如下目录：

```
/sys/devices/platform/backlight1/backlight/backlight1
```

此目录下的文件如图 24.15.1.3 所示：

```
root@ATK-DLRK356X:/sys/devices/platform/backlight1/backlight/backlight1# ls
actual_brightness  brightness  max_brightness  subsystem  uevent
bl_power           device      power           type
root@ATK-DLRK356X:/sys/devices/platform/backlight1/backlight/backlight1#
```

图 24.15.1.3 目录下的文件和子目录

图 24.15.1.3 中的 brightness 表示当前亮度等级，max_bgigntness 表示最大亮度等级。当前这两个文件内容如图 24.15.1.4 所示：（出厂内核已经将 brightness 配置为 255，最亮等级）

```
root@ATK-DLRK356X:/sys/devices/platform/backlight1/backlight/backlight1# cat max_brightness
255
root@ATK-DLRK356X:/sys/devices/platform/backlight1/backlight/backlight1# cat brightness
200
root@ATK-DLRK356X:/sys/devices/platform/backlight1/backlight/backlight1#
```

图 24.15.1.4 brightness 和 max_brightness 文件内容

从图 24.15.1.4 可以看出，当前屏幕亮度等级为 200，根据前面的分析可以，这个约等于 78% 亮度，屏幕最大亮度等级为 255。如果我们要修改屏幕亮度，只需要向 brightness 写入需要设置的屏幕亮度等级即可。比如设置屏幕亮度等级为 10，那么可以使用如下命令：

```
echo 10 > brightness
```

输入上述命令以后就会发现屏幕亮度变暗了，如果设置 brightness 为 0 的话就会关闭 LCD 背光，屏幕就会熄灭。

24.15.2 MIPI 协议实测

1、协议数据采集

前面我们详细讲解了 MIPI DSI LP 协议，从理论上理解了 MIPI 屏幕初始化原理。但是在实际的工作中，可能遇到屏幕驱动不正常，这个时候就可以使用逻辑分析仪直接抓取相关波形，看看真实的波形数据对不对。这里我们主要抓取 MIPI DSI LP 阶段的波形，也就是 Lanes 数据线上的 data0。这里使用正点原子逻辑分析仪，如图 24.15.3.1 所示：



图 24.15.3.1 正点原子逻辑分析仪

设置好逻辑分析仪，选择 MIPI DSI LP 协议，然后将逻辑分析仪的相关采集线接到开发板的 Data0 交叉线上。注意，如果开发板上 Data0 不好焊接采集线，那可以刮掉开发板上相关信号线的阻焊层，露出信号新铜皮，然后直接焊接上去。建议在产品调试阶段，给 MIPI DSI 的 Data0 交叉线留出测试点，方便后续测试。

设置好逻辑分析仪的协议以及信号线通道，如图 24.15.3.2 所示：



图 24.15.3.2 逻辑分析仪协议设置

协议设置好以后还要进行设备设置，如图 24.15.3.3 所示：

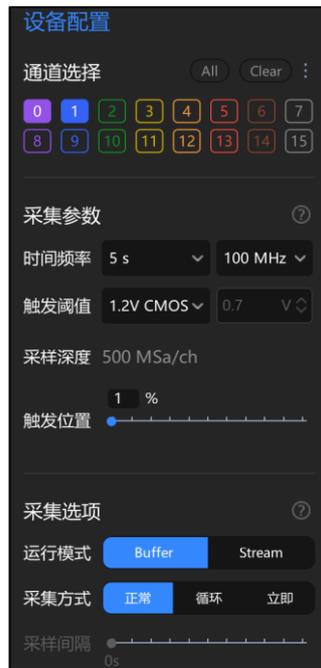


图 24.15.3.3 设备设置

图 24.15.3.3 中采集时间 5S，这个时间要大家根据自己的时间情况估算一下，比如我们的开发板从上电到 MIPI 屏幕初始化完成肯定要不了 5S，所以这里设置 5S。采集频率为 100MHz，这个够了，因为 MIPI DSI LP 协议速度为 10M。MIPI DSI LP 协议是 1.2V 的，触发电压为 0.7V。

设置好以后就可以采集波形，步骤如下：

- 1)、按着开发板复位按键不要松手。
- 2)、点击逻辑分析仪上位机软件开始按钮，开始采集。
- 3)、松开开发板的复位按钮，系统正常启动，这个过程就会初始化 MIPI 屏幕，那么初始化过程的数据就会被逻辑分析仪采集到。

采集完成以后就会显示采集到的波形，如图 24.15.3.4 所示：

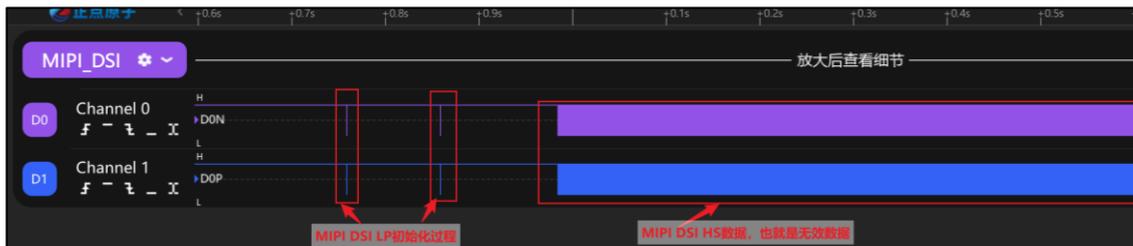


图 24.15.3.4 采集到的波形数据

图 24.15.3.4 中，前面两个竖线就是我们采集到的 MIPI DSI LP 协议数据，最后面那一大段数据就是屏幕正常工作的图像数据，对于我们本章实验来说是无效数据，不用管这部分。我们接下来就详细分析实际采集到的 MIPI DSI LP 协议数据。

2、协议分析

首先分析第一帧数据，如图 24.15.3.5 所示：

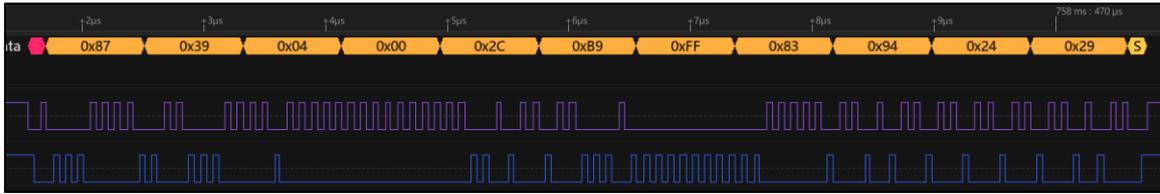


图 24.15.3.5 第一帧数据

图 24.15.3.5 中最开始红色的地方为 ESC，也就是 Escape，如图 24.15.3.6 所示：



图 24.15.3.6 ESC 头部

图 24.15.3.5 中第一帧数据就是我们在示例代码 24.14.3.1 中第 25 行就是 MIPI 屏幕初始化的第一行数据：

39 00 04 B9 FF 83 94

- 0X39：表示此命令是 DCS 长写命令。
- 0X00：此命令发送后延时 0ms。
- 0X04：此命令负载长度为 4 个字节。
- 0XB9~0X94：具体的负载数据。

图 24.15.3.5 中采集到的实际数据是按照前面说的长度包打包好的数据，很明显这一帧是长包数据，总结一下这一帧的数据内容如下：

Escape 0X87 0X39 0X04 0X00 0X2C 0XB9 0XFF 0X83 0X94 0X24 0X29

按照前面讲解的长包数据结构分析上面采集的数据，结果如图 24.15.3.7 所示：

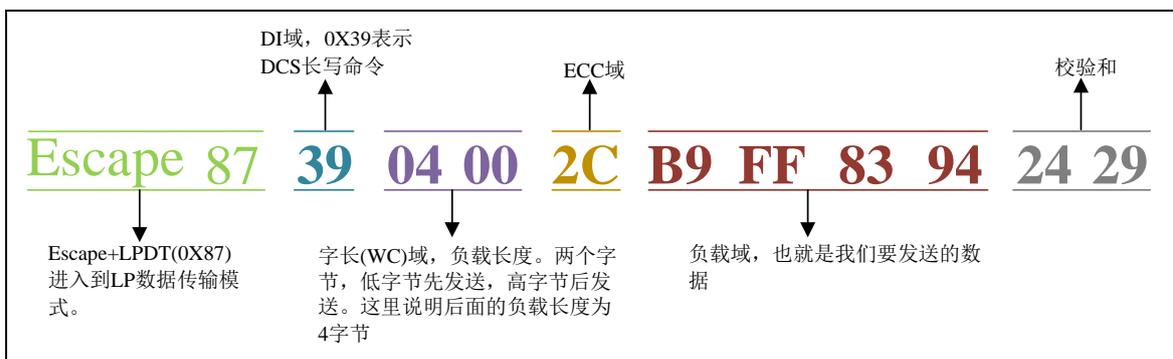


图 24.15.3.7 数据帧长包结构解析

图 24.15.3.7 就是对图 24.15.3.5 中实际采集到的数据进行结构分析结果，这是一个标准的长数据包，前面的 Escape+0X87 就是进入到数据传输模式。后面的数据就是标准的长数据包格式了。

接下来再看一个短数据包结构，示例代码 24.14.3.8 中第 26 行就是一个短数据包，内容如下：

15 00 02 36 01

- 0X15：表示这是个 DCS 短写指令。
- 0X00：此命令发送后延时 0ms。

0X02: 命令负载长度为 2 个字节。

0X36~0X01: 具体的负载数据。

采集到的实际波形如图 24.15.3.8 所示:

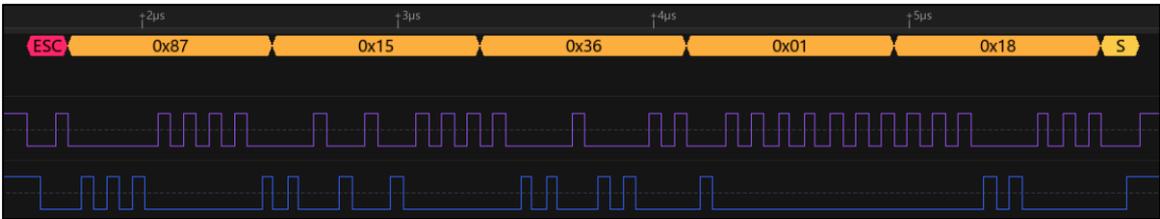


图 24.15.3.8 短数据包波形

同样使用短数据包对图 24.15.3.8 中的波形进行分析, 结果如图 24.15.3.9 所示:

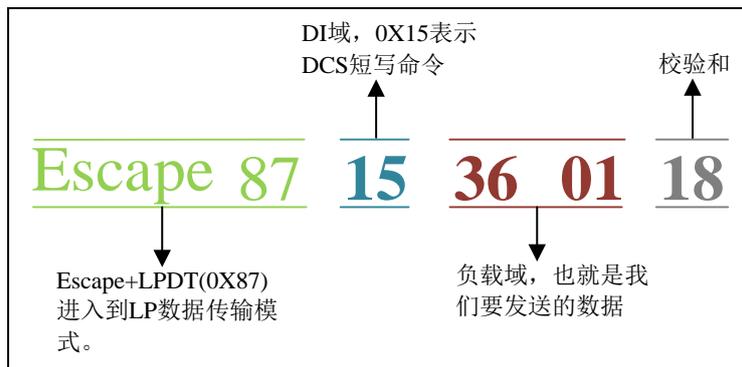


图 24.15.3.9 数据帧短包结构解析

最后我们看一下数据帧延时, 在我们实际采集的波形中会发现 LP 模式有两段, 如图 24.15.3.10 所示:

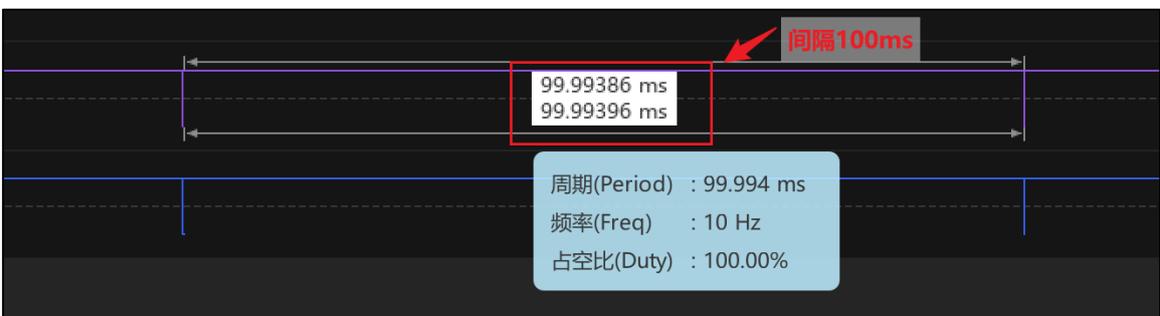


图 24.15.3.10 数据发送延时

从图 24.15.3.10 可以看出, 这两段数据帧之间的延时是 100ms。这个就是示例代码 24.14.3.8 中第 47 行发送命令设置的延时时间, 命令内容如下:

```
05 64 01 11
```

0X05: 表示这是个 DCS 短写命令(无参数)。

0X64: 此命令发送以后延时 0X64=100ms。

0X01: 此命令负载长度为 1 个字节。

0X11: 实际发送的负载数据。

所以, 图 24.15.3.10 中的 100ms 延时, 就是这条命令要求的。

至此, 使用逻辑分析仪抓取 MIPI DSI LP 波形并进行协议分析就讲解完了, 我们分析了长短数据包, 包括命令延时。大家在实际的 MIPI 屏幕调试中就可以直接使用逻辑分析仪抓取波形来分析屏幕初始化是否正确了。

第二十五章 HDMI 屏幕驱动实验

第二十六章 LVDS 屏幕驱动实验

第二十七章 EDP 屏幕驱动实验

第二十八章 Linux I2C 驱动实验

对于 I2C 我相信大家都很熟悉,基本上做过单片机开发的朋友都接触过,在电子产品硬件设计当中,I2C 是一种很常见的同步、串行、低速、近距离通信接口,用于连接各种 IC、传感器等器件,它们都会提供 I2C 接口与 SoC 主控相连,比如陀螺仪、加速度计、触摸屏等,其最大优势在于可以在总线上扩展多个外围设备的支持。

Linux 内核开发者为了让驱动开发工程师在内核中方便的添加自己的 I2C 设备驱动程序,更容易的在 linux 下驱动自己的 I2C 接口硬件,进而引入了 I2C 总线框架。与 Linux 下的 platform 虚拟总线不同的是,I2C 是实际的物理总线,所以 I2C 总线框架也是 Linux 下总线、设备、驱动模型的产物。

本章我们来学习一下如何在 Linux 下的 I2C 总线框架,以及如何使用 I2C 总线框架编写一个 I2C 接口的外设驱动程序;本章重点是学习 Linux 下的 I2C 总线框架。

28.1 I2C& AP3216C 简介

28.1.1 I2C 简介

I2C 是很常见的一种总线协议，I2C 是 NXP 公司设计的，I2C 使用两条线在主控制器和从机之间进行数据通信。一条是 SCL(串行时钟线)，另外一条是 SDA(串行数据线)，这两条数据线需要接上拉电阻，总线空闲的时候 SCL 和 SDA 处于高电平。I2C 总线标准模式下速度可以达到 100Kb/S，快速模式下可以达到 400Kb/S。I2C 总线工作是按照一定的协议来运行的，接下来就看一下 I2C 协议。

I2C 是支持多从机的，也就是一个 I2C 控制器下可以挂多个 I2C 从设备，这些不同的 I2C 从设备有不同的器件地址，这样 I2C 主控制器就可以通过 I2C 设备的器件地址访问指定的 I2C 设备了，一个 I2C 总线连接多个 I2C 设备如图 28.1.1.1 所示：

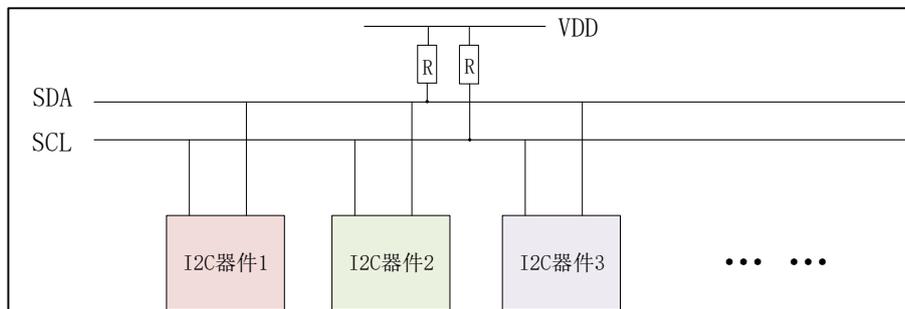


图 28.1.1.1 I2C 多个设备连接结构图

图 28.1.1.1 中 SDA 和 SCL 这两根线必须要接一个上拉电阻，一般是 4.7K。其余的 I2C 从器件都挂接到 SDA 和 SCL 这两根线上，这样就可以通过 SDA 和 SCL 这两根线来访问多个 I2C 设备。

接下来看一下 I2C 协议有关的术语：

1、起始位

顾名思义，也就是 I2C 通信起始标志，通过这个起始位就可以告诉 I2C 从机，“我”要开始进行 I2C 通信了。在 SCL 为高电平的时候，SDA 出现下降沿就表示为起始位，如图 28.1.1.2 所示：

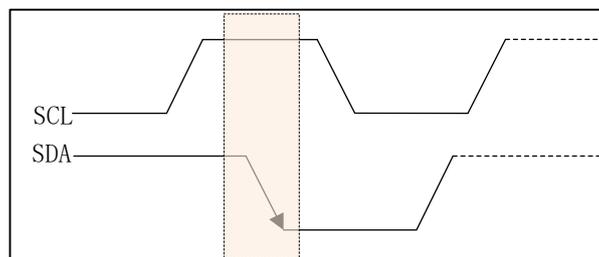


图 28.1.1.1.2 I2C 通信起始位

2、停止位

停止位就是停止 I2C 通信的标志位，和起始位的功能相反。在 SCL 位高电平的时候，SDA 出现上升沿就表示为停止位，如图 28.1.1.3 所示：

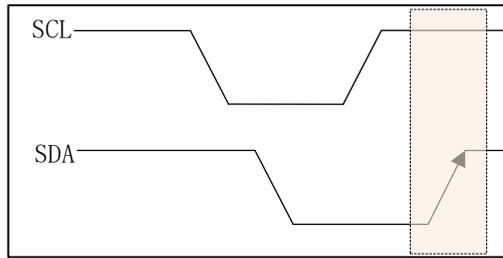


图 28.1.1.3 I2C 通信停止位

3、数据传输

I2C 总线在数据传输的时候要保证在 SCL 高电平期间，SDA 上的数据稳定，因此 SDA 上的数据变化只能在 SCL 低电平期间发生，如图 28.1.1.4 所示：

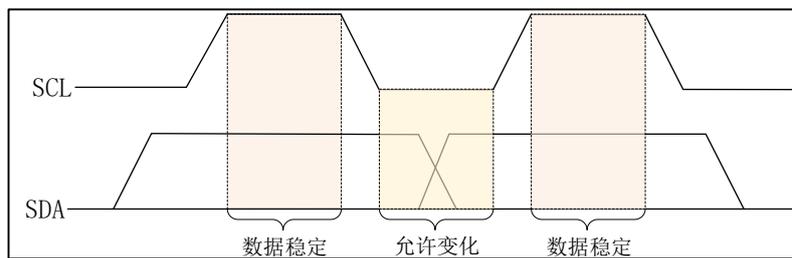


图 28.1.1.4 I2C 数据传输

4、应答信号

当 I2C 主机发送完 8 位数据以后会将 SDA 设置为输入状态，等待 I2C 从机应答，也就是等待 I2C 从机告诉主机它接收到数据了。应答信号是由从机发出的，主机需要提供应答信号所需的时钟，主机发送完 8 位数据以后紧跟着的一个时钟信号就是给应答信号使用的。从机通过将 SDA 拉低来表示发出应答信号，表示通信成功，否则表示通信失败。

5、I2C 写时序

主机通过 I2C 总线与从机之间进行通信不外乎两个操作：写和读，I2C 总线单字节写时序如图 28.1.1.5 所示：

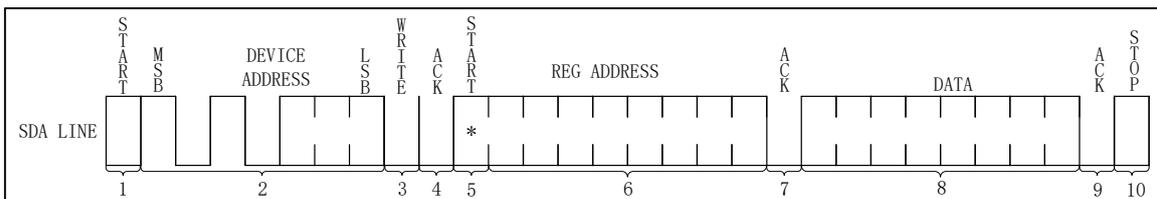


图 28.1.1.5 I2C 写时序

图 28.1.1.5 就是 I2C 写时序，我们来看一下写时序的具体步骤：

- 1)、开始信号。
- 2)、发送 I2C 设备地址，每个 I2C 器件都有一个设备地址，通过发送具体的设备地址来决定访问哪个 I2C 器件。这是一个 8 位的数据，其中高 7 位是设备地址，最后 1 位是读写位，为 1 的话表示这是一个读操作，为 0 的话表示这是一个写操作。
- 3)、I2C 器件地址后面跟着一个读写位，为 0 表示写操作，为 1 表示读操作。
- 4)、从机发送的 ACK 应答信号。
- 5)、重新发送开始信号。
- 6)、发送要写写入数据的寄存器地址。

- 7)、从机发送的 ACK 应答信号。
- 8)、发送要写入寄存器的数据。
- 9)、从机发送的 ACK 应答信号。
- 10)、停止信号。

6、I2C 读时序

I2C 总线单字节读时序如图 28.1.1.6 所示:

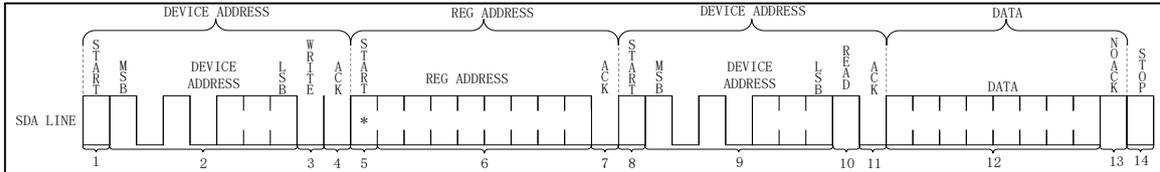


图 28.1.1.6 I2C 单字节读时序

I2C 单字节读时序比写时序要复杂一点，读时序分为 4 大步，第一步是发送设备地址，第二步是发送要读取的寄存器地址，第三步重新发送设备地址，最后一步就是 I2C 从器件输出要读取的寄存器值，我们具体来看一下这步。

- 1)、主机发送起始信号。
- 2)、主机发送要读取的 I2C 从设备地址。
- 3)、读写控制位，因为是向 I2C 从设备发送数据，因此是写信号。
- 4)、从机发送的 ACK 应答信号。
- 5)、重新发送 START 信号。
- 6)、主机发送要读取的寄存器地址。
- 7)、从机发送的 ACK 应答信号。
- 8)、重新发送 START 信号。
- 9)、重新发送要读取的 I2C 从设备地址。
- 10)、读写控制位，这里是读信号，表示接下来是从 I2C 从设备里面读取数据。
- 11)、从机发送的 ACK 应答信号。
- 12)、从 I2C 器件里面读取到的数据。
- 13)、主机发出 NO ACK 信号，表示读取完成，不需要从机再发送 ACK 信号了。
- 14)、主机发出 STOP 信号，停止 I2C 通信。

7、I2C 多字节读写时序

有时候我们需要读写多个字节，多字节读写时序和单字节的基本一致，只是在读写数据的时候可以连续发送多个自己的数据，其他的控制时序都是和单字节一样的。

28.1.2 RK3568 I2C 简介

RK3568 支持 6 个独立 I2C: I2C0、I2C1、I2C2、I2C3、I2C4、I2C5。I2C 控制器支持以下特性:

- ① 兼容 i2c 总线
- ② AMBA APB 从接口
- ③ 支持 I2C 总线主模式
- ④ 软件可编程时钟频率和传输速率高达 400Kbit/sec
- ⑤ 支持 7 位和 10 位寻址模式

- ⑥ 中断或轮询驱动的多字节数据传输
- ⑦ 时钟拉伸和等待状态生成
- ⑧ 过滤掉 SCL 和 SDA 上的故障

关于 RK3568 IIC 更多详细的介绍，路径：[开发板光盘](#) → [03、核心板资料](#) → [核心板板载芯片资料](#) → [Rockchip RK3568 TRM Part1 V1.1-20210301.pdf](#) (RK3568 参考手册 1) .pdf 和 [Rockchip RK3568 TRM Part2 V1.1-20210301](#) (RK3568 参考手册 2) .pdf。

28.1.3 AP3216C 简介

ATK-DLRK3568 开发板上通过 I2C5 连接了一个三合一环境传感器：AP3216C，AP3216C 是由敦南科技推出的一款传感器，其支持环境光强度(ALS)、接近距离(PS)和红外线强度(IR)这三个环境参数检测。该芯片可以通过 IIC 接口与主控制相连，并且支持中断，AP3216C 的特点如下：

- ①、I2C 接口，快速模式下波特率可以到 400Kbit/S
- ②、多种工作模式选择：ALS、PS+IR、ALS+PS+IR、PD 等等。
- ③、内建温度补偿电路。
- ④、宽工作温度范围(-30° C ~ +80° C)。
- ⑤、超小封装，4.1mm x 2.4mm x 1.35mm
- ⑥、环境光传感器具有 16 位分辨率。
- ⑦、接近传感器和红外传感器具有 10 位分辨率。

AP3216C 常被用于手机、平板、导航设备等，其内置的接近传感器可以用于检测是否有物体接近，比如手机上用来检测耳朵是否接触听筒，如果检测到的话就表示正在打电话，手机就会关闭手机屏幕以省电。也可以使用环境光传感器检测光照强度，可以实现自动背光亮度调节。AP3216C 结构如图 28.1.3.1 所示：

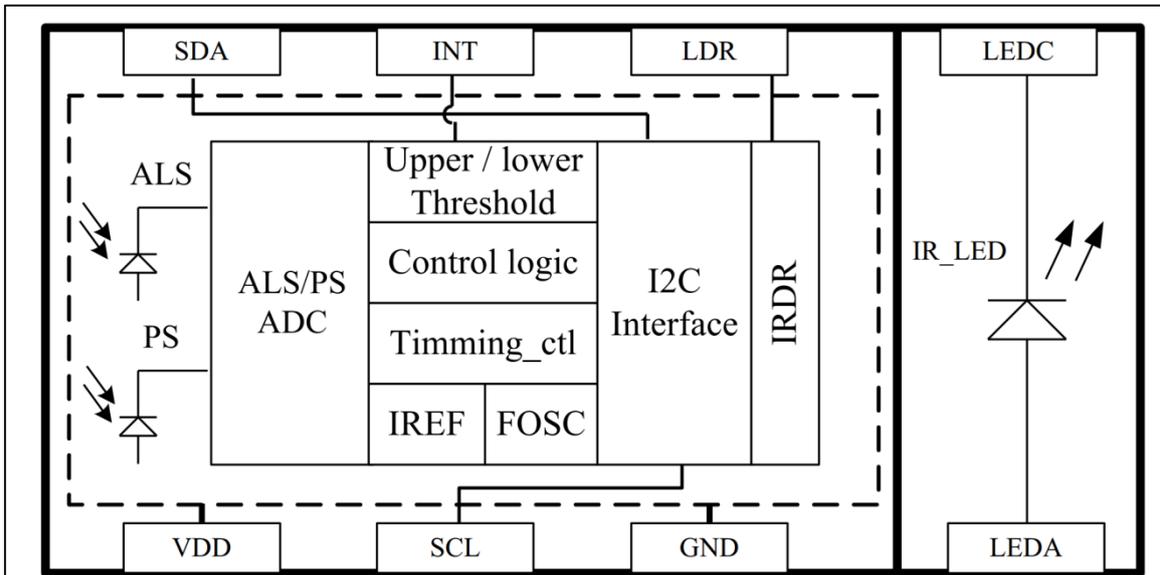


图 28.1.3.1 AP3216C 结构图

AP3216 的设备地址为 0X1E，同几乎所有的 I2C 从器件一样，AP3216C 内部也有一些寄存器，通过这些寄存器我们可以配置 AP3216C 的工作模式，并且读取相应的数据。AP3216C 我们用的寄存器如表 28.1.3.1 所示：

寄存器地址	位	寄存器功能	描述
-------	---	-------	----

0X00	2:0	系统模式	000: 掉电模式(默认)。 001: 使能 ALS。 010: 使能 PS+IR。 011: 使能 ALS+PS+IR。 100: 软复位。 101: ALS 单次模式。 110: PS+IR 单次模式。 111: ALS+PS+IR 单次模式。
0X0A	7	IR 低位数据	0: IR&PS 数据有效, 1:无效
	1:0		IR 最低 2 位数据。
0X0B	7:0	IR 高位数据	IR 高 8 位数据。
0X0C	7:0	ALS 低位数据	ALS 低 8 位数据。
0X0D	7:0	ALS 高位数据	ALS 高 8 位数据。
0X0E	7	PS 低位数据	0, 物体在远离; 1, 物体在接近。
	6		0, IR&PS 数据有效; 1, IR&PS 数据无效
	3:0		PS 最低 4 位数据。
0X0F	7	PS 高位数据	0, 物体在远离; 1, 物体在接近。
	6		0, IR&PS 数据有效; 1, IR&PS 数据无效
	5:0		PS 最低 6 位数据。

表 28.1.3.1 本章使用的 AP3216C 寄存器表

在表 28.1.3.1 中, 0X00 这个寄存器是模式控制寄存器, 用来设置 AP3216C 的工作模式, 一般开始先将其设置为 0X04, 也就是先软件复位一次 AP3216C。接下来根据实际使用情况选择合适的工作模式, 比如设置为 0X03, 也就是开启 ALS+PS+IR。0X0A~0X0F 这 6 个寄存器就是数据寄存器, 保存着 ALS、PS 和 IR 这三个传感器获取到的数据值。如果同时打开 ALS、PS 和 IR 的读取间隔最少要 112.5ms, 因为 AP3216C 完成一次转换需要 112.5ms。关于 AP3216C 的介绍就到这里, 如果要想详细的研究此芯片的话, 请大家自行查阅其数据手册。

28.2 Linux I2C 总线框架简介

使用裸机的方式编写一个 I2C 器件的驱动程序, 我们一般要实现两部分:

- ①、I2C 主机驱动。
- ②、I2C 设备驱动。

I2C 主机驱动也就是 SoC 的 I2C 控制器对应的驱动程序, I2C 设备驱动其实就是挂在 I2C 总线下的具体设备对应的驱动程序, 例如 eeprom、触摸屏 IC、传感器 IC 等; 对于主机驱动来说, 一旦编写完成就不需要再做修改, 其他的 I2C 设备直接调用主机驱动提供的 API 函数完成读写操作即可。这个正好符合 Linux 的驱动分离与分层的思想, 因此 Linux 内核也将 I2C 驱动分为两部分。

Linux 内核开发者为了让驱动开发工程师在内核中方便的添加自己的 I2C 设备驱动程序, 方便大家更容易的在 linux 下驱动自己的 I2C 接口硬件, 进而引入了 I2C 总线框架, 我们一般也叫作 I2C 子系统, Linux 下 I2C 子系统总体框架如下所示:

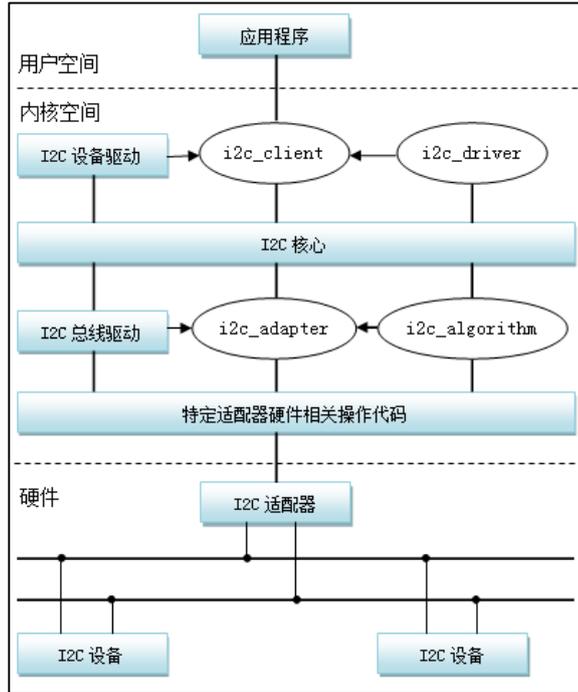


图 28.2.3.1 I2C 子系统框架图

从图 28.2.3.1 可以知道，I2C 子系统分为三大组成部分：

1、I2C 核心(I2C-core)

I2C 核心提供了 I2C 总线驱动（适配器）和设备驱动的注册、注销方法，I2C 通信方法 (algorithm)与具体硬件无关的代码，以及探测设备地址的上层代码等；

2、I2C 总线驱动(I2C adapter)

I2C 总线驱动是 I2C 适配器的软件实现，提供 I2C 适配器与从设备间完成数据通信的能力。I2C 总线驱动由 i2c_adapter 和 i2c_algorithm 来描述。I2C 适配器是 SoC 中内置 i2c 控制器的软件抽象，可以理解为他所代表的是一个 I2C 主机；

3、I2C 设备驱动(I2C client driver)

包括两部分：设备的注册和驱动的注册。

I2C 子系统帮助内核统一管理 I2C 设备，让驱动开发工程师在内核中可以更加容易地添加自己的 I2C 设备驱动程序。

28.2.1 I2C 总线驱动

首先来看一下 I2C 总线，在讲 platform 的时候就说过，platform 是虚拟出来的一条总线，目的是为了实线总线、设备、驱动框架。对于 I2C 而言，不需要虚拟出一条总线，直接使用 I2C 总线即可。I2C 总线驱动重点是 I2C 适配器(也就是 SoC 的 I2C 接口控制器)驱动，这里要用到两个重要的数据结构：i2c_adapter 和 i2c_algorithm，I2C 子系统将 SoC 的 I2C 适配器(控制器)抽象成一个 i2c_adapter 结构体，i2c_adapter 结构体定义在 include/linux/i2c.h 文件中，结构体内容如下：

示例代码 28.2.1 i2c_adapter 结构体

```
672 struct i2c_adapter {
673     struct module *owner;
```

```

674     unsigned int class;          /* classes to allow probing for */
675     const struct i2c_algorithm *algo; /* the algorithm to access the
bus */
676     void *algo_data;
677
678     /* data fields that are valid for all devices */
679     const struct i2c_lock_operations *lock_ops;
680     struct rt_mutex bus_lock;
681     struct rt_mutex mux_lock;
682
683     int timeout;                /* in jiffies */
684     int retries;
685     struct device dev;         /* the adapter device */
686
687     int nr;
688     char name[48];
689     struct completion dev_released;
690
691     struct mutex userspace_clients_lock;
692     struct list_head userspace_clients;
693
694     struct i2c_bus_recovery_info *bus_recovery_info;
695     const struct i2c_adapter_quirks *quirks;
696
697     struct irq_domain *host_notify_domain;
698 };
    
```

第 675 行, `i2c_algorithm` 类型的指针变量 `algo`, 对于一个 I2C 适配器, 肯定要对外提供读写 API 函数, 设备驱动程序可以使用这些 API 函数来完成读写操作。`i2c_algorithm` 就是 I2C 适配器与 IIC 设备进行通信的方法。

`i2c_algorithm` 结构体定义在 `include/linux/i2c.h` 文件中, 内容如下:

示例代码 28.2.1.2 `i2c_algorithm` 结构体

```

526 struct i2c_algorithm {
527     /*
528      * If an adapter algorithm can't do I2C-level access, set
529      * master_xfer to NULL. If an adapter algorithm can do SMBus
530      * access, set smbus_xfer. If set to NULL, the SMBus protocol is
531      * simulated using common I2C messages.
532      *
533      * master_xfer should return the number of messages successfully
534      * processed, or a negative value on error
535      */
536     int (*master_xfer)(struct i2c_adapter *adap,
                        struct i2c_msg *msgs,
    
```

```

537         int num);
538     int (*master_xfer_atomic)(struct i2c_adapter *adap,
539                             struct i2c_msg *msgs, int num);
540     int (*smbus_xfer)(struct i2c_adapter *adap, u16 addr,
541                     unsigned short flags, char read_write,
542                     u8 command, int size, union i2c_smbus_data *data);
543     int (*smbus_xfer_atomic)(struct i2c_adapter *adap, u16 addr,
544                             unsigned short flags, char read_write,
545                             u8 command, int size, union i2c_smbus_data *data);
546
547     /* To determine what the adapter supports */
548     u32 (*functionality)(struct i2c_adapter *adap);
549
550 #if IS_ENABLED(CONFIG_I2C_SLAVE)
551     int (*reg_slave)(struct i2c_client *client);
552     int (*unreg_slave)(struct i2c_client *client);
553 #endif
554 };

519 struct i2c_algorithm {
520     /* If an adapter algorithm can't do I2C-level access, set
521     master_xfer
522     to NULL. If an adapter algorithm can do SMBus access, set
523     smbus_xfer. If set to NULL, the SMBus protocol is simulated
524     using common I2C messages */
525     /* master_xfer should return the number of messages successfully
526     processed, or a negative value on error */
527     int (*master_xfer)(struct i2c_adapter *adap, struct i2c_msg
528     *msgs,
529                     int num);
530     int (*smbus_xfer)(struct i2c_adapter *adap, u16 addr,
531                     unsigned short flags, char read_write,
532                     u8 command, int size, union i2c_smbus_data *data);
533
534     /* To determine what the adapter supports */
535     u32 (*functionality)(struct i2c_adapter *);
536
537 #if IS_ENABLED(CONFIG_I2C_SLAVE)
538     int (*reg_slave)(struct i2c_client *client);
539     int (*unreg_slave)(struct i2c_client *client);
540 #endif
541 };
    
```

第 526 行, `master_xfer` 就是 I2C 适配器的传输函数, 可以通过此函数来完成与 IIC 设备之间的通信。

第 528 行, `smbus_xfer` 就是 SMBUS 总线的传输函数。smbus 协议是从 I2C 协议的基础上发展而来的, 他们之间有很大的相似度, SMBus 与 I2C 总线之间在时序特性上存在一些差别, 应用于移动 PC 和桌面 PC 系统中的低速率通讯。

综上所述, I2C 总线驱动, 或者说 I2C 适配器驱动的主要工作就是初始化 `i2c_adapter` 结构体变量, 然后设置 `i2c_algorithm` 中的 `master_xfer` 函数。完成以后通过 `i2c_add_numbered_adapter` 或 `i2c_add_adapter` 这两个函数向 I2C 子系统注册设置好的 `i2c_adapter`, 这两个函数的原型如下:

```
int i2c_add_adapter(struct i2c_adapter *adapter)
int i2c_add_numbered_adapter(struct i2c_adapter *adap)
```

这两个函数的区别在于 `i2c_add_adapter` 会动态分配一个总线编号, 而 `i2c_add_numbered_adapter` 函数则指定一个静态的总线编号。函数参数和返回值含义如下:

adapter 或 adap: 要添加到 Linux 内核中的 `i2c_adapter`, 也就是 I2C 适配器。

返回值: 0, 成功; 负值, 失败。

如果要删除 I2C 适配器的话使用 `i2c_del_adapter` 函数即可, 函数原型如下:

```
void i2c_del_adapter(struct i2c_adapter * adap)
```

函数参数和返回值含义如下:

adap: 要删除的 I2C 适配器。

返回值: 无。

关于 I2C 的总线(控制器或适配器)驱动就讲解到这里, 一般 SoC 的 I2C 总线驱动都是由半导体厂商编写的, 比如 RK3568 的 I2C 适配器驱动 RK 官方已经编写好了, 这个不需要用户去编写。因此 I2C 总线驱动对我们这些 SoC 使用者来说是被屏蔽掉的, 我们只要专注于 I2C 设备驱动即可, 除非你是在半导体公司上班, 工作内容就是写 I2C 适配器驱动。

28.2.2 I2C 总线设备

I2C 设备驱动重点关注两个数据结构: `i2c_client` 和 `i2c_driver`, 根据总线、设备和驱动模型, I2C 总线上一小节已经讲了。还剩下设备和驱动, `i2c_client` 用于描述 I2C 总线下的设备, `i2c_driver` 则用于描述 I2C 总线下的设备驱动, 类似于 platform 总线下的 `platform_device` 和 `platform_driver`。

1、i2c_client 结构体

`i2c_client` 结构体定义在 `include/linux/i2c.h` 文件中, 内容如下:

示例代码 28.2.2.1 `i2c_client` 结构体

```
328 struct i2c_client {
329     unsigned short flags;          /* div., see below */
330     unsigned short addr;          /* chip address - NOTE: 7bit */
331     /* addresses are stored in the */
332     /* _LOWER_ 7 bits */
333     char name[I2C_NAME_SIZE];
334     struct i2c_adapter *adapter;  /* the adapter we sit on */
335     struct device dev;           /* the device structure */
336     int init_irq;                /* irq set at initialization */
337     int irq;                      /* irq issued by device */
```

```

338     struct list_head detected;
339 #if IS_ENABLED(CONFIG_I2C_SLAVE)
340     i2c_slave_cb_t slave_cb;    /* callback for slave mode */
341 #endif
342 };
    
```

一个 I2C 设备对应一个 `i2c_client` 结构体变量，系统每检测到一个 I2C 从设备就会给这个设备分配一个 `i2c_client`。

2、i2c_driver 结构体

`i2c_driver` 类似 `platform_driver`，是我们编写 I2C 设备驱动重点要处理的内容，`i2c_driver` 结构体定义在 `include/linux/i2c.h` 文件中，内容如下：

示例代码 28.2.2.2 i2c_driver 结构体

```

267 struct i2c_driver {
268     unsigned int class;
269
270     /* Standard driver model interfaces */
271     int (*probe)(struct i2c_client *, const struct i2c_device_id *);
272     int (*remove)(struct i2c_client *);
273
274     /* New driver model interface to aid the seamless removal of the
275      * current probe()'s, more commonly unused than used second
parameter.
276      */
277     int (*probe_new)(struct i2c_client *);
278
279     /* driver model interfaces that don't relate to enumeration */
280     void (*shutdown)(struct i2c_client *);
281
282     /* Alert callback, for example for the SMBus alert protocol.
283      * The format and meaning of the data value depends on the
protocol.
284      * For the SMBus alert protocol, there is a single bit of data
passed
285      * as the alert response's low bit ("event flag").
286      * For the SMBus Host Notify protocol, the data corresponds to
the
287      * 16-bit payload data reported by the slave device acting as
master.
288      */
289     void (*alert)(struct i2c_client *, enum i2c_alert_protocol
protocol,
290                 unsigned int data);
291
    
```

```

292     /* a ioctl like command that can be used to perform specific
functions
293     * with the device.
294     */
295     int (*command)(struct i2c_client *client, unsigned int cmd, void
*arg);
296
297     struct device_driver driver;
298     const struct i2c_device_id *id_table;
299
300     /* Device detection callback for automatic device creation */
301     int (*detect)(struct i2c_client *, struct i2c_board_info *);
302     const unsigned short *address_list;
303     struct list_head clients;
304
305     bool disable_i2c_core_irq_mapping;
306 };
307 #define to_i2c_driver(d) container_of(d, struct i2c_driver, driver)
    
```

第 271 行, 当 I2C 设备和驱动匹配成功以后 probe 函数就会执行, 和 platform 驱动一样。

第 297 行, device_driver 驱动结构体, 如果使用设备树的话, 需要设置 device_driver 的 of_match_table 成员变量, 也就是驱动的兼容(compatible)属性。

第 298 行, id_table 是传统的、未使用设备树的设备匹配 ID 表。

对于我们 I2C 设备驱动编写人来说, 重点工作就是构建 i2c_driver, 构建完成以后需要向 I2C 子系统注册这个 i2c_driver。i2c_driver 注册函数为 int i2c_register_driver, 此函数原型如下:

```

int i2c_register_driver(struct module      *owner,
                      struct i2c_driver *driver)
    
```

函数参数和返回值含义如下:

owner: 一般为 THIS_MODULE。

driver: 要注册的 i2c_driver。

返回值: 0, 成功; 负值, 失败。

另外 i2c_add_driver 也常常用于注册 i2c_driver, i2c_add_driver 是一个宏, 定义如下:

示例代码 28.2.2.3 i2c_add_driver 宏

```

806     #define i2c_add_driver(driver) \
807         i2c_register_driver(THIS_MODULE, driver)
    
```

i2c_add_driver 就是对 i2c_register_driver 做了一个简单的封装, 只有一个参数, 就是要注册的 i2c_driver。

注销 I2C 设备驱动的时候需要将前面注册的 i2c_driver 从 I2C 子系统中注销掉, 需要用到 i2c_del_driver 函数, 此函数原型如下:

```

void i2c_del_driver(struct i2c_driver *driver)
    
```

函数参数和返回值含义如下:

driver: 要注销的 i2c_driver。

返回值: 无。

i2c_driver 的注册示例代码如下:

示例代码 28.2.2.4 i2c_driver 注册流程

```

1  /* i2c 驱动的 probe 函数 */
2  static int xxx_probe(struct i2c_client *client,
                       const struct i2c_device_id *id)
3  {
4      /* 函数具体程序 */
5      return 0;
6  }
7
8  /* i2c 驱动的 remove 函数 */
9  static int ap3216c_remove(struct i2c_client *client)
10 {
11     /* 函数具体程序 */
12     return 0;
13 }
14
15 /* 传统匹配方式 ID 列表 */
16 static const struct i2c_device_id xxx_id[] = {
17     {"xxx", 0},
18     {}
19 };
20
21 /* 设备树匹配列表 */
22 static const struct of_device_id xxx_of_match[] = {
23     { .compatible = "xxx" },
24     { /* Sentinel */ }
25 };
26
27 /* i2c 驱动结构体 */
28 static struct i2c_driver xxx_driver = {
29     .probe = xxx_probe,
30     .remove = xxx_remove,
31     .driver = {
32         .owner = THIS_MODULE,
33         .name = "xxx",
34         .of_match_table = xxx_of_match,
35     },
36     .id_table = xxx_id,
37 };
38
39 /* 驱动入口函数 */
40 static int __init xxx_init(void)
41 {
    
```

```

42     int ret = 0;
43
44     ret = i2c_add_driver(&xxx_driver);
45     return ret;
46 }
47
48 /* 驱动出口函数 */
49 static void __exit xxx_exit(void)
50 {
51     i2c_del_driver(&xxx_driver);
52 }
53
54 module_init(xxx_init);
55 module_exit(xxx_exit);
    
```

第 16~19 行, `i2c_device_id`, 无设备树的时候匹配 ID 表。

第 22~25 行, `of_device_id`, 设备树所使用的匹配表。

第 28~37 行, `i2c_driver`, 当 I2C 设备和 I2C 驱动匹配成功以后 `probe` 函数就会执行, 这些和 `platform` 驱动一样, `probe` 函数里面基本就是标准的字符设备驱动那一套了。

28.2.3 I2C 设备和驱动匹配过程

I2C 设备和驱动的匹配过程是由 I2C 子系统核心层来完成的, `drivers/i2c/i2c-core-base.c` 就是 I2C 的核心部分, I2C 核心提供了一些与具体硬件无关的 API 函数, 比如前面讲过的:

1、i2c_adapter 注册/注销函数

```

int i2c_add_adapter(struct i2c_adapter *adapter)
int i2c_add_numbered_adapter(struct i2c_adapter *adap)
void i2c_del_adapter(struct i2c_adapter * adap)
    
```

2、i2c_driver 注册/注销函数

```

int i2c_register_driver(struct module *owner, struct i2c_driver *driver)
int i2c_add_driver (struct i2c_driver *driver)
void i2c_del_driver(struct i2c_driver *driver)
    
```

设备和驱动的匹配过程也是由核心层完成的, I2C 总线的数据结构为 `i2c_bus_type`, 定义在 `drivers/i2c/i2c-core-base.c` 文件, `i2c_bus_type` 内容如下:

示例代码 28.2.3.1 `i2c_bus_type` 结构体

```

505 struct bus_type i2c_bus_type = {
506     .name      = "i2c",
507     .match     = i2c_device_match,
508     .probe    = i2c_device_probe,
509     .remove   = i2c_device_remove,
510     .shutdown = i2c_device_shutdown,
511 };
    
```

`.match` 就是 I2C 总线的设备和驱动匹配函数, 在这里就是 `i2c_device_match` 这个函数, 此函数内容如下:

示例代码 28.2.3.2 i2c_device_match 函数

```

103 static int i2c_device_match(struct device *dev,
                               struct device_driver *drv)
104 {
105     struct i2c_client *client = i2c_verify_client(dev);
106     struct i2c_driver *driver;
107
108
109     /* Attempt an OF style match */
110     if (i2c_of_match_device(drv->of_match_table, client))
111         return 1;
112
113     /* Then ACPI style match */
114     if (acpi_driver_match_device(dev, drv))
115         return 1;
116
117     driver = to_i2c_driver(drv);
118
119     /* Finally an I2C match */
120     if (i2c_match_id(driver->id_table, client))
121         return 1;
122
123     return 0;
124 }
    
```

第 110 行, `i2c_of_match_device` 函数用于完成设备树中定义的设备与驱动匹配过程。比较 I2C 设备节点的 `compatible` 属性和 `of_device_id` 中的 `compatible` 属性是否相等, 如果相当的话就表示 I2C 设备和驱动匹配。

第 114 行, `acpi_driver_match_device` 函数用于 ACPI 形式的匹配。

第 120 行, `i2c_match_id` 函数用于传统的、无设备树的 I2C 设备和驱动匹配过程。比较 I2C 设备名字和 `i2c_device_id` 的 `name` 字段是否相等, 相等的话就说明 I2C 设备和驱动匹配成功。

28.3 RK3568 I2C 适配器驱动分析

上一小节我们讲解了 Linux 下的 I2C 子系统, 重点分为 I2C 适配器驱动和 I2C 设备驱动, 其中 I2C 适配器驱动就是 SoC 的 I2C 控制器驱动。I2C 设备驱动是需要用户根据不同的 I2C 从设备去编写, 而 I2C 适配器驱动一般都是 SoC 厂商去编写的, 比如 RK 就已经提供了 RK3568 的 I2C 适配器驱动程序。在内核源码 `arch/arm64/boot/dts/rockchip/rk3568.dtsi` 设备树文件中找到 RK3568 的 I2C 控制器节点, 节点内容如下所示:

示例代码 41.3.1 I2C1 控制器节点

```

2900 i2c1: i2c@fe5a0000 {
2901     compatible = "rockchip,rk3399-i2c";
2902     reg = <0x0 0xfe5a0000 0x0 0x1000>;
2903     clocks = <&cru CLK_I2C1>, <&cru PCLK_I2C1>;
2904     clock-names = "i2c", "pclk";
    
```

```

2905     interrupts = <GIC_SPI 47 IRQ_TYPE_LEVEL_HIGH>;
2906     pinctrl-names = "default";
2907     pinctrl-0 = <&i2c1_xfer>;
2908     #address-cells = <1>;
2909     #size-cells = <0>;
2910     status = "disabled";
2911 };
    
```

重点关注 i2c1 节点的 compatible 属性值，因为通过 compatible 属性值可以在 Linux 源码里面找到对应的驱动文件。这里 i2c1 节点的 compatible 属性值“rockchip,rk3399-i2c”，在 Linux 源码中搜索这个字符串即可找到对应的驱动文件。RK3568 的 I2C 适配器驱动驱动文件为 drivers/i2c/busses/i2c-rk3x.c，在此文件中有如下内容：

示例代码 28.3.2 i2c-rk3x.c 文件代码段

```

1258 static const struct of_device_id rk3x_i2c_match[] = {
1259     {
1260         .compatible = "rockchip,rv1108-i2c",
1261         .data = &rv1108_soc_data
1262     },
1263     {
1264         .compatible = "rockchip,rv1126-i2c",
1265         .data = &rv1126_soc_data
1266     },
1267     {
1268         .compatible = "rockchip,rk3066-i2c",
1269         .data = &rk3066_soc_data
1270     },
1271     {
1272         .compatible = "rockchip,rk3188-i2c",
1273         .data = &rk3188_soc_data
1274     },
1275     {
1276         .compatible = "rockchip,rk3228-i2c",
1277         .data = &rk3228_soc_data
1278     },
1279     {
1280         .compatible = "rockchip,rk3288-i2c",
1281         .data = &rk3288_soc_data
1282     },
1283     {
1284         .compatible = "rockchip,rk3399-i2c",
1285         .data = &rk3399_soc_data
1286     },
1287     {},
1288 };
    
```

从示例代码 28.3.2 可以看出, STM32MP1 的 I2C 适配器驱动是个标准的 platform 驱动, 由此可以看出, 虽然 I2C 总线为别的设备提供了一种总线驱动框架, 但是 I2C 适配器却是 platform 驱动。就像你的部门老大是你的领导, 你是他的下属, 但是放到整个公司, 你的部门老大却也是老板的下属。

当设备和驱动匹配成功以后 rk3x_i2c_probe 函数就会执行, rk3x_i2c_probe 函数就会完成 I2C 适配器初始化工作。

rk3x_i2c_probe 函数内容如下所示(有省略):

示例代码 28.3.3 rk3x_i2c_probe 函数代码段

```

1291 static int rk3x_i2c_probe(struct platform_device *pdev)
1292 {
1293     struct device_node *np = pdev->dev.of_node;
1294     const struct of_device_id *match;
1295     struct rk3x_i2c *i2c;
1296     struct resource *mem;
1297     int ret = 0;
1298     u32 value;
1299     int irq;
1300     unsigned long clk_rate;
1301
1302     i2c = devm_kzalloc(&pdev->dev, sizeof(struct rk3x_i2c),
GFP_KERNEL);
1303     if (!i2c)
1304         return -ENOMEM;
1305
1306     match = of_match_node(rk3x_i2c_match, np);
1307     i2c->soc_data = match->data;
1308
1309     /* use common interface to get I2C timing properties */
1310     i2c_parse_fw_timings(&pdev->dev, &i2c->t, true);
1311
1312     strncpy(i2c->adap.name, "rk3x-i2c", sizeof(i2c->adap.name));
1313     i2c->adap.owner = THIS_MODULE;
1314     i2c->adap.algo = &rk3x_i2c_algorithm;
1315     i2c->adap.retries = 3;
1316     i2c->adap.dev.of_node = np;
1317     i2c->adap.algo_data = i2c;
1318     i2c->adap.dev.parent = &pdev->dev;
1319
1320     i2c->dev = &pdev->dev;
1321
1322     spin_lock_init(&i2c->lock);
1323     init_waitqueue_head(&i2c->wait);
1324

```

```

1325     i2c->i2c_restart_nb.notifier_call = rk3x_i2c_restart_notify;
1326     i2c->i2c_restart_nb.priority = 128;
1327     ret = register_pre_restart_handler(&i2c->i2c_restart_nb);
1328     if (ret) {
1329         dev_err(&pdev->dev, "failed to setup i2c restart
handler.\n");
1330         return ret;
1331     }
1332
1333     mem = platform_get_resource(pdev, IORESOURCE_MEM, 0);
1334     i2c->regs = devm_ioremap_resource(&pdev->dev, mem);
1335     if (IS_ERR(i2c->regs))
1336         return PTR_ERR(i2c->regs);
1337
.....
1376
1377     /* IRQ setup */
1378     irq = platform_get_irq(pdev, 0);
1379     if (irq < 0) {
1380         dev_err(&pdev->dev, "cannot find rk3x IRQ\n");
1381         return irq;
1382     }
1383
1384     ret = devm_request_irq(&pdev->dev, irq, rk3x_i2c_irq,
1385                          0, dev_name(&pdev->dev), i2c);
1386     if (ret < 0) {
1387         dev_err(&pdev->dev, "cannot request IRQ\n");
1388         return ret;
1389     }
1390
1391     platform_set_drvdata(pdev, i2c);
1392
1393     if (i2c->soc_data->calc_timings == rk3x_i2c_v0_calc_timings)
1394     {
1395         /* Only one clock to use for bus clock and peripheral
clock */
1396         i2c->clk = devm_clk_get(&pdev->dev, NULL);
1397         i2c->pclk = i2c->clk;
1398     } else {
1399         i2c->clk = devm_clk_get(&pdev->dev, "i2c");
1400         i2c->pclk = devm_clk_get(&pdev->dev, "pclk");
1401     }

```

```

1402     if (IS_ERR(i2c->clk)) {
1403         ret = PTR_ERR(i2c->clk);
1404         if (ret != -EPROBE_DEFER)
1405             dev_err(&pdev->dev, "Can't get bus clk: %d\n",
ret);
1406         return ret;
1407     }
1408     if (IS_ERR(i2c->pclk)) {
1409         ret = PTR_ERR(i2c->pclk);
1410         if (ret != -EPROBE_DEFER)
1411             dev_err(&pdev->dev, "Can't get periph
clk: %d\n", ret);
1412         return ret;
1413     }
1414
1415     ret = clk_prepare(i2c->clk);
1416     if (ret < 0) {
1417         dev_err(&pdev->dev, "Can't prepare bus clk: %d\n",
ret);
1418         return ret;
1419     }
1420     ret = clk_prepare(i2c->pclk);
1421     if (ret < 0) {
1422         dev_err(&pdev->dev, "Can't prepare periph
clock: %d\n", ret);
1423         goto err_clk;
1424     }
1425
1426     i2c->clk_rate_nb.notifier_call = rk3x_i2c_clk_notifier_cb;
1427     ret = clk_notifier_register(i2c->clk, &i2c->clk_rate_nb);
1428     if (ret != 0) {
1429         dev_err(&pdev->dev, "Unable to register clock
notifier\n");
1430         goto err_pclk;
1431     }
1432
1433     clk_rate = clk_get_rate(i2c->clk);
1434     rk3x_i2c_adapt_div(i2c, clk_rate);
1435
1436     ret = i2c_add_adapter(&i2c->adap);
1437     if (ret < 0)
.....
1449 }
    
```

第 1302 行, RK 使用 `rk3x_i2c` 结构体来表示 RK 系列 SOC 的 I2C 控制器, 这里使用 `devm_kzalloc` 函数来申请内存。

第 1310 行, 调用 `drivers/i2c/i2c-core-base.c` 下的 `i2c_parse_fw_timings` 函数设置 I2C 频率, 如果不设置 “clock-frequency” 则使用默认值 100KHZ, 如果设备树节点设置了 “clock-frequency” 属性的话 I2C 频率就使用 `clock-frequency` 属性值。

第 1314 行, `rk3x_i2c` 结构体有个 `adap` 的成员变量, `adap` 就是 `i2c_adapter`, 这里初始化 `i2c_adapter`。同时设置 `i2c_adapter` 的 `algo` 成员变量为 `rk3x_i2c_algorithm`, 也就是设置 `i2c_algorithm`。

第 1333~1334 行, 调用 `platform_get_resource` 函数从设备树中获取 I2C1 控制器寄存器物理基地址, 也就是 `0xfe5a0000`。获取到寄存器基地址以后使用 `devm_ioremap_resource` 函数对其进行内存映射, 得到可以在 Linux 中使用的虚拟地址。

第 1378 行, 调用 `platform_get_irq` 函数获取中断号。

第 1384 行, 注册 I2C 控制器的中断。

第 1436 行, 调用 `i2c_add_adapter` 函数向 Linux 内核注册 `i2c_adapter`。

`rk3x_i2c_probe` 函数主要的工作就是一下两点:

- ①、初始化 `i2c_adapter`, 设置 `i2c_algorithm` 为 `rk3x_i2c_algorithm`, 最后向 Linux 内核注册 `i2c_adapter`。
- ②、初始化 I2C1 控制器的相关寄存器。`rk3x_i2c_algorithm` 包含 I2C1 适配器与 I2C 设备的通信函数 `master_xfer`, `rk3x_i2c_algorithm` 结构体定义如下:

示例代码 28.3.4 `rk3x_i2c_algorithm` 结构体

```
1218 static const struct i2c_algorithm rk3x_i2c_algorithm = {
1219     .master_xfer          = rk3x_i2c_xfer,
1220     .functionality       = rk3x_i2c_func,
1221 };
```

我们先来看一下 `functionality`, `functionality` 用于返回此 I2C 适配器支持什么样的通信协议, 在这里 `functionality` 就是 `rk3x_i2c_func` 函数, `rk3x_i2c_func` 函数内容如下:

示例代码 28.3.5 `rk3x_i2c_func` 函数

```
1213 static u32 rk3x_i2c_func(struct i2c_adapter *adap)
1214 {
1215     return I2C_FUNC_I2C | I2C_FUNC_SMBUS_EMUL |
1216     I2C_FUNC_PROTOCOL_MANGLING;
1216 }
```

重点来看一下 `rk3x_i2c_xfer` 函数, 因为最终就是通过此函数来完成与 I2C 设备通信的, 此函数内容如下:

示例代码 28.3.6 `rk3x_i2c_xfer` 函数

```
1071 static int rk3x_i2c_xfer(struct i2c_adapter *adap,
1072                          struct i2c_msg *msgs, int num)
1073 {
1074     struct rk3x_i2c *i2c = (struct rk3x_i2c *)adap->algo_data;
1075     unsigned long timeout, flags;
1076     u32 val;
1077     int ret = 0;
1078     int i;
```

```

1079
1080     if (i2c->suspended)
1081         return -EACCES;
1082
1083     spin_lock_irqsave(&i2c->lock, flags);
1084
1085     clk_enable(i2c->clk);
1086     clk_enable(i2c->pclk);
1087
1088     i2c->is_last_msg = false;
1089
1090     /*
1091      * Process msgs. We can handle more than one message at once
1092      * (see
1093      * rk3x_i2c_setup()).
1094      */
1095     for (i = 0; i < num; i += ret) {
1096         ret = rk3x_i2c_setup(i2c, msgs + i, num - i);
1097
1098         if (ret < 0) {
1099             dev_err(i2c->dev, "rk3x_i2c_setup()
1100 failed\n");
1101             break;
1102         }
1103
1104         if (i + ret >= num)
1105             i2c->is_last_msg = true;
1106
1107         rk3x_i2c_start(i2c);
1108
1109         spin_unlock_irqrestore(&i2c->lock, flags);
1110
1111         timeout = wait_event_timeout(i2c->wait, !i2c->busy,
1112 msecs_to_jiffies(WAIT_TIMEOUT));
1113
1114         spin_lock_irqsave(&i2c->lock, flags);
1115
1116         if (timeout == 0) {
1117             dev_err(i2c->dev, "timeout, ipd: 0x%02x,
1118 state: %d\n",
1119                 i2c_readl(i2c, REG_IPD), i2c->state);
1120         }
1121     }

```

```

1118         /* Force a STOP condition without interrupt */
1119         rk3x_i2c_disable_irq(i2c);
1120         val = i2c_readl(i2c, REG_CON) &
REG_CON_TUNING_MASK;
1121         val |= REG_CON_EN | REG_CON_STOP;
1122         i2c_writel(i2c, val, REG_CON);
1123
1124         i2c->state = STATE_IDLE;
1125
1126         ret = -ETIMEDOUT;
1127         break;
1128     }
1129
1130     if (i2c->error) {
1131         ret = i2c->error;
1132         break;
1133     }
1134 }
1135
1136 rk3x_i2c_disable_irq(i2c);
1137 rk3x_i2c_disable(i2c);
1138
1139 clk_disable(i2c->pclk);
1140 clk_disable(i2c->clk);
1141
1142 spin_unlock_irqrestore(&i2c->lock, flags);
1143
1144 return ret < 0 ? ret : num;
1145 }
    
```

第 1095 行调用 `rk3x_i2c_setup` 函数来进行 I2C 相关的配置和准备工作。如果返回值 `ret` 小于 0，表示设置失败，函数会打印错误消息并跳出循环。

第 1105 行，启动函数来启动 I2C 传输，具体内容这里就不分析了，大家感兴趣可以自己阅读代码。

28.4 I2C 设备驱动编写流程

I2C 适配器驱动 SOC 厂商已经替我们编写好了，我们需要做的就是编写具体的设备驱动，本小节我们就来学习一下 I2C 设备驱动的详细编写流程。

28.4.1 I2C 设备信息描述

1、未使用设备树的时候

首先肯定要描述 I2C 设备节点信息，先来看一下没有使用设备树的时候是如何在 BSP 里面描述 I2C 设备信息的，在未使用设备树的时候需要在 BSP 里面使用 `i2c_board_info` 结构体来描

述一个具体的 I2C 设备。i2c_board_info 结构体如下:

示例代码 28.4.1.1 i2c_board_info 结构体

```

415 struct i2c_board_info {
416     char          type[I2C_NAME_SIZE];
417     unsigned short flags;
418     unsigned short addr;
419     const char    *dev_name;
420     void          *platform_data;
421     struct device_node *of_node;
422     struct fwnode_handle *fwnode;
423     const struct property_entry *properties;
424     const struct resource *resources;
425     unsigned int  num_resources;
426     int          irq;
427 };
    
```

type 和 addr 这两个成员变量是必须要设置的, 一个是 I2C 设备的名字, 一个是 I2C 设备的器件地址。举个例子, 打开 arch/arm/mach-imx/mach-armadillo5x0.c 文件, 此文件中有关于 s35390a 这个 I2C 器件对应的设备描述信息:

示例代码 28.4.1.2 s35390a 的 I2C 设备信息

```

260 static struct i2c_board_info armadillo5x0_i2c_rtc = {
261     I2C_BOARD_INFO("s35390a", 0x30),
262 };
    
```

示例代码 28.4.1.2 中使用 I2C_BOARD_INFO 来完成 armadillo5x0_i2c_rtc 的初始化工作, I2C_BOARD_INFO 是一个宏, 定义如下:

示例代码 28.4.1.3 I2C_BOARD_INFO 宏

```

439 #define I2C_BOARD_INFO(dev_type, dev_addr) \
440     .type = dev_type, .addr = (dev_addr)
    
```

可以看出, I2C_BOARD_INFO 宏其实就是设置 i2c_board_info 的 type 和 addr 这两个成员变量, 因此示例代码 28.4.1.2 的主要工作就是设置 I2C 设备名字为 s35390a, 器件地址为 0X30。

大家可以在 Linux 源码里面全局搜索 i2c_board_info, 会找到大量以 i2c_board_info 定义的 I2C 设备信息, 这些就是未使用设备树的时候 I2C 设备的描述方式, 当采用了设备树以后就不会再使用 i2c_board_info 来描述 I2C 设备了。

2、使用设备树的时候

使用设备树的时候 I2C 设备信息通过创建相应的节点就行了, 比如在我们的 ATK-DLRK3568 的开发板上有一个 I2C 器件 AP3216C, 这是三合一的环境传感器, 并且该器件挂在 RK3568 I2C5 总线接口上, 因此必须在 i2c5 节点下创建一个子节点来描述 AP3216C 设备, 节点示例如下所示:

示例代码 28.4.1.4 i2c 从设备节点示例

```

1  &i2c5 {
2      status = "okay";
3      ap3216c@1e {
4          compatible = "alientek,ap3216c";
5          reg = <0x1e>;
    
```

```
6     };
7 };
```

第 2 行, 设置 i2c5 的 status 为 “okay”。

第 3~6 行, 向 i2c5 添加 ap3216c 子节点, 第 3 行 “ap3216c@1e” 是子节点名字, “@” 后面的 “1e” 就是 ap3216c 的 I2C 器件地址。第 4 行设置 compatible 属性值为 “alientek,ap3216c”。第 5 行的 reg 属性也是设置 ap3216c 的器件地址的, 因此值为 0x1e。I2C 设备节点的创建重点是 compatible 属性和 reg 属性的设置, 一个用于匹配驱动, 一个用于设置器件地址。

25.4.2 I2C 设备数据收发处理流程

在 25.2.2 小节已经说过了, I2C 设备驱动首先要做的就是初始化 i2c_driver 并向 Linux 内核注册。当设备和驱动匹配以后 i2c_driver 里面的 probe 函数就会执行, probe 函数里面所做的就是字符设备驱动那一套了。一般需要在 probe 函数里面初始化 I2C 设备, 要初始化 I2C 设备就必须能够对 I2C 设备寄存器进行读写操作, 这里就要用到 i2c_transfer 函数了。i2c_transfer 函数最终会调用 I2C 适配器中 i2c_algorithm 里面的 master_xfer 函数, 对于 RK3568 而言就是 rk3x_i2c_xfer 这个函数。i2c_transfer 函数原型如下:

```
int i2c_transfer(struct i2c_adapter *adap,
                 struct i2c_msg     *msgs,
                 int                 num)
```

函数参数和返回值含义如下:

adap: 所使用的 I2C 适配器, i2c_client 会保存其对应的 i2c_adapter。

msgs: I2C 要发送的一个或多个消息。

num: 消息数量, 也就是 msgs 的数量。

返回值: 负值, 失败, 其他非负值, 发送的 msgs 数量。

我们重点来看一下 msgs 这个参数, 这是一个 i2c_msg 类型的指针参数, I2C 进行数据收发说白了就是消息的传递, Linux 内核使用 i2c_msg 结构体来描述一个消息。i2c_msg 结构体定义在 include/uapi/linux/i2c.h 文件中, 结构体内容如下:

示例代码 28.4.2.1 i2c_msg 结构体

```
69 struct i2c_msg {
70     __u16 addr;    /* 从机地址 */
71     __u16 flags;  /* 标志 */
72 #define I2C_M_RD                0x0001
73
74 #define I2C_M_TEN                0x0010
75 #define I2C_M_DMA_SAFE          0x0200
76
77
78 #define I2C_M_RECV_LEN          0x0400
79 #define I2C_M_NO_RD_ACK        0x0800
80 #define I2C_M_IGNORE_NAK       0x1000
81 #define I2C_M_REV_DIR_ADDR     0x2000
82 #define I2C_M_NOSTART          0x4000
83 #define I2C_M_STOP              0x8000
84     __u16 len;    /* 消息(本 msg)长度 */
```

```

85     __u8 *buf;      /* 消息数据 */
86 };
    
```

使用 `i2c_transfer` 函数发送数据之前要先构建好 `i2c_msg`, 使用 `i2c_transfer` 进行 I2C 数据收发 的示例代码如下:

示例代码 28.4.2.2 I2C 设备多寄存器数据读写

```

1  /* 设备结构体 */
2  struct xxx_dev {
3      .....
4      void *private_data; /* 私有数据, 一般会设置为 i2c_client */
5  };
6
7  /*
8   * @description      : 读取 I2C 设备多个寄存器数据
9   * @param - dev      : I2C 设备
10 * @param - reg       : 要读取的寄存器首地址
11 * @param - val       : 读取到的数据
12 * @param - len       : 要读取的数据长度
13 * @return            : 操作结果
14 */
15 static int xxx_read_regs(struct xxx_dev *dev, u8 reg, void *val,
16                          int len)
17 {
18     int ret;
19     struct i2c_msg msg[2];
20     struct i2c_client *client = (struct i2c_client *)
21                                 dev->private_data;
22
23     /* msg[0], 第一条写消息, 发送要读取的寄存器首地址 */
24     msg[0].addr = client->addr;      /* I2C 器件地址 */
25     msg[0].flags = 0;                /* 标记为发送数据 */
26     msg[0].buf = &reg;               /* 读取的首地址 */
27     msg[0].len = 1;                  /* reg 长度 */
28
29     /* msg[1], 第二条读消息, 读取寄存器数据 */
30     msg[1].addr = client->addr;      /* I2C 器件地址 */
31     msg[1].flags = I2C_M_RD;        /* 标记为读取数据 */
32     msg[1].buf = val;                /* 读取数据缓冲区 */
33     msg[1].len = len;                /* 要读取的数据长度 */
34
35     ret = i2c_transfer(client->adapter, msg, 2);
36     if(ret == 2) {
37         ret = 0;
38     } else {
    
```

```

37     ret = -EREMOTEIO;
38 }
39 return ret;
40 }
41
42 /*
43 * @description      : 向 I2C 设备多个寄存器写入数据
44 * @param - dev      : 要写入的设备结构体
45 * @param - reg      : 要写入的寄存器首地址
46 * @param - val      : 要写入的数据缓冲区
47 * @param - len      : 要写入的数据长度
48 * @return           : 操作结果
49 */
50 static s32 xxx_write_regs(struct xxx_dev *dev, u8 reg, u8 *buf,
                           u8 len)
51 {
52     u8 b[256];
53     struct i2c_msg msg;
54     struct i2c_client *client = (struct i2c_client *)
                                   dev->private_data;
55
56     b[0] = reg;                /* 寄存器首地址 */
57     memcpy(&b[1], buf, len);   /* 将要发送的数据拷贝到数组 b 里面 */
58
59     msg.addr = client->addr;    /* I2C 器件地址 */
60     msg.flags = 0;            /* 标记为写数据 */
61
62     msg.buf = b;              /* 要发送的数据缓冲区 */
63     msg.len = len + 1;        /* 要发送的数据长度 */
64
65     return i2c_transfer(client->adapter, &msg, 1);
66 }
    
```

第 2~5 行, 设备结构体, 在设备结构体里面添加一个执行 void 的指针成员变量 `private_data`, 此成员变量用于保存设备的私有数据。在 I2C 设备驱动中我们一般将其指向 I2C 设备对应的 `i2c_client`。

第 15~40 行, `xxx_read_regs` 函数用于读取 I2C 设备多个寄存器数据。第 18 行定义了一个 `i2c_msg` 数组, 2 个数组元素, 因为 I2C 读取数据的时候要先发送要读取的寄存器地址, 然后再读取数据, 所以需要准备两个 `i2c_msg`。一个用于发送寄存器地址, 一个用于读取寄存器值。对于 `msg[0]`, 将 `flags` 设置为 0, 表示写数据。`msg[0]` 的 `addr` 是 I2C 设备的器件地址, `msg[0]` 的 `buf` 成员变量就是要读取的寄存器地址。对于 `msg[1]`, 将 `flags` 设置为 `I2C_M_RD`, 表示读取数据。`msg[1]` 的 `buf` 成员变量用于保存读取到的数据, `len` 成员变量就是要读取的数据长度。调用 `i2c_transfer` 函数完成 I2C 数据读操作。

第 50~66 行, `xxx_write_regs` 函数用于向 I2C 设备多个寄存器写数据, I2C 写操作要比读操作简单一点, 因此一个 `i2c_msg` 即可。数组 `b` 用于存放寄存器首地址和要发送的数据, 第 59 行设置 `msg` 的 `addr` 为 I2C 器件地址。第 60 行设置 `msg` 的 `flags` 为 0, 也就是写数据。第 62 行设置要发送的数据, 也就是数组 `b`。第 63 行设置 `msg` 的 `len` 为 `len+1`, 因为要加上一个字节的寄存器地址。最后通过 `i2c_transfer` 函数完成向 I2C 设备的写操作。

另外还有两个 API 函数分别用于 I2C 数据的收发操作, 这两个函数最终都会调用 `i2c_transfer`。首先来看一下 I2C 数据发送函数 `i2c_master_send`, 函数原型如下:

```
int i2c_master_send(const struct i2c_client *client,
                   const char *buf,
                   int count)
```

函数参数和返回值含义如下:

client: I2C 设备对应的 `i2c_client`。

buf: 要发送的数据。

count: 要发送的数据字节数, 要小于 64KB, 以为 `i2c_msg` 的 `len` 成员变量是一个 `u16`(无符号 16 位)类型的数据。

返回值: 负值, 失败, 其他非负值, 发送的字节数。

I2C 数据接收函数为 `i2c_master_recv`, 函数原型如下:

```
int i2c_master_recv(const struct i2c_client *client,
                   char *buf,
                   int count)
```

函数参数和返回值含义如下:

client: I2C 设备对应的 `i2c_client`。

buf: 要接收的数据。

count: 要接收的数据字节数, 要小于 64KB, 以为 `i2c_msg` 的 `len` 成员变量是一个 `u16`(无符号 16 位)类型的数据。

返回值: 负值, 失败, 其他非负值, 发送的字节数。

关于 Linux 下 I2C 设备驱动的编写流程就讲解到这里, 重点就是 `i2c_msg` 的构建和 `i2c_transfer` 函数的调用, 接下来我们就编写 AP3216C 这个 I2C 设备的 Linux 驱动。

28.5 硬件原理图分析

AP3612C 原理图如图 28.5.1 所示:

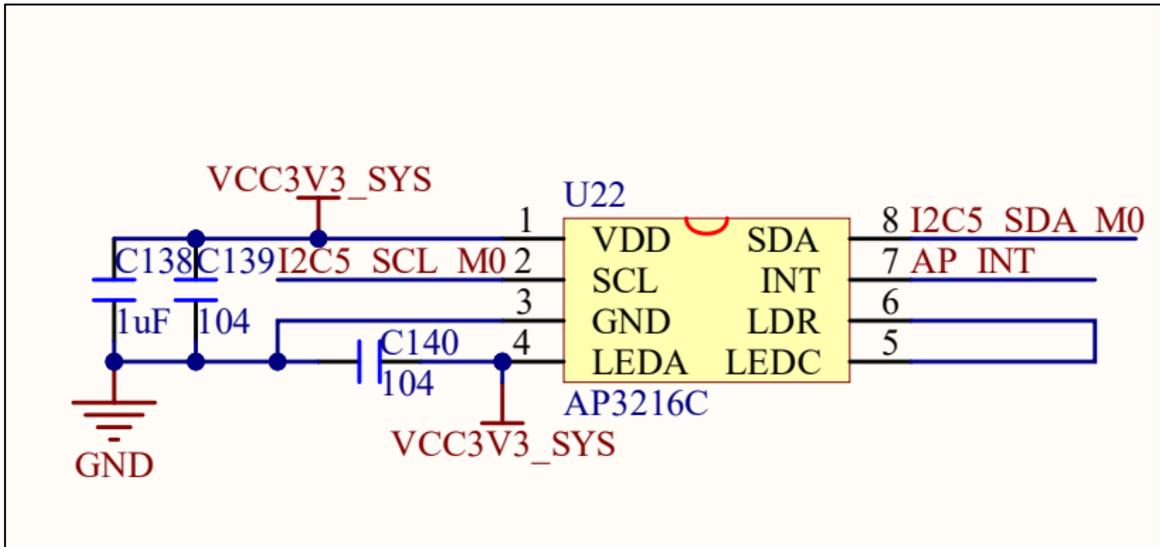


图 28.5.1 AP3216C 原理图

从图 28.5.1 可以看出 AP3216C 使用的是 I2C5，AP3216C 还有个中断引脚，这里我们没有用到中断功能。

28.6 实验程序编写

本实验对应的例程路径为：[开发板光盘](#)→1、[程序源码](#)→2、[Linux 驱动例程](#)→18_iic。

28.6.1 修改设备树

1、注释出厂配置的 AP3216C

AP3216C 用到了 I2C5 接口，出厂系统已经使用了内核自带的驱动，此时若要编写驱动，应该注释自带的驱动设备树配置。打开 rk3568-atk-evb1-ddr4-v10.dtsi，然后找到如下内容：

示例代码 28.6.1.1 I2C5 的 pinmux 配置

```

635     ap3216c_ls@1e {
636         compatible = "ls_ap321xx";
637         reg = <0x1e>;
638         type = <SENSOR_TYPE_LIGHT>;
639         //irq-gpio = <&gpio4 RK_PD2 IRQ_TYPE_EDGE_FALLING>;
640         irq_enable = <0>;
641         poll_delay_ms = <20>;
642         layout = <1>;
643         status = "disabled";
644     };
645
646     ap3216c_ps@1e {
647         compatible = "ps_ap321xx";
648         reg = <0x1e>;
649         type = <SENSOR_TYPE_PROXIMITY>;
650         //irq-gpio = <&gpio4 RK_PD2 IRQ_TYPE_EDGE_FALLING>;
    
```

```

651         irq_enable = <0>;
652         poll_delay_ms = <20>;
653         layout = <1>;
654         status = "disabled";
655     };
    
```

示例代码 28.6.1.1 中, 将 643 行改的 status 改为“disabled”, 第 654 行同理也改为“disabled”这样自带的 AP3216C 驱动将不会生效。

2、在 i2c5 节点追加 ap3216c 子节点

接着我们在 rk3568-atk-evb1-ddr4-v10.dtsi 文件, 通过节点内容追加的方式, 向 i2c5 节点中添加“ap3216c@1e”子节点, 节点如下所示:

示例代码 28.6.1.2 向 i2c5 追加 ap3216c 子节点

```

1     &i2c5 {
2         ap3216c@1e {
3             compatible = "alientek,ap3216c";
4             reg = <0x1e>;
5         };
6     };
    
```

第 2 行, ap3216c 子节点, @后面的“1e”是 ap3216c 的器件地址。

第 3 行, 设置 compatible 值为“alientek,ap3216c”。

第 4 行, reg 属性也是设置 ap3216c 器件地址的, 因此 reg 设置为 0x1e。

设备树修改完成以后使用“/build.sh kernel”重新编译一下, 然后重新烧写 boot.img 启动 Linux 内核。/sys/bus/i2c/devices 目录下存放着所有 I2C 设备, 如果设备树修改正确的话, 会在 /sys/bus/i2c/devices 目录下看到一个名为“5-001e”的子目录, 如图 28.6.1.1 所示:

```

root@ATK-DLRK356X:/# cd /sys/bus/i2c/devices
root@ATK-DLRK356X:/sys/bus/i2c/devices# ls
0-001c 1-0014 4-001a-1 5-0036 5-0051 i2c-1 i2c-4 i2c-6
0-0020 4-001a 5-001e 5-0036-1 i2c-0 i2c-3 i2c-5
root@ATK-DLRK356X:/sys/bus/i2c/devices#
    
```

图 28.6.1.1 当前系统 I2C 设备

图 28.6.1.1 中的“5-001e”就是 ap3216c 的设备目录, “1e”就是 ap3216c 器件地址。进入 5-001e 目录, 可以看到“name”文件, name 文件保存着此设备名字, 在这里就是“ap3216c”, 如图 28.6.1.2 所示:

```

root@ATK-DLRK356X:/sys/bus/i2c/devices# cd 5-001e
root@ATK-DLRK356X:/sys/bus/i2c/devices/5-001e# cat name
ap3216c
root@ATK-DLRK356X:/sys/bus/i2c/devices/5-001e#
    
```

图 28.6.1.2 ap3216c 器件名字

28.6.2 AP3216C 驱动编写

新建名为“18_iic”的文件夹, 然后在 21_iic 文件夹里面创建 vscode 工程, 工作区命名为“iic”。工程创建好以后新建 ap3216c.c 和 ap3216creg.h 这两个文件, ap3216c.c 为 AP3216C 的驱动代码, ap3216creg.h 是 AP3216C 寄存器头文件。先在 ap3216creg.h 中定义好 AP3216C 的寄存器, 输入如下内容,

示例代码 28.6.2.1 ap3216creg.h 文件代码段

```

1  #ifndef AP3216C_H
2  #define AP3216C_H
3  /*****
4  Copyright © ALIENTEK Co., Ltd. 1998-2029. All rights reserved.
5  文件名      : ap3216creg.h
6  作者        : 正点原子 Linux 团队
7  版本        : V1.0
8  描述        : AP3216C 寄存器地址描述头文件
9  其他        : 无
10 论坛        : www.openedv.com
11 日志        : 初版 V1.0 2021/03/19 正点原子 Linux 团队创建
12 *****/
13
14 #define AP3216C_ADDR          0X1E          /* AP3216C 器件地址      */
15
16 /* AP3316C 寄存器 */
17 #define AP3216C_SYSTEMCONG  0x00          /* 配置寄存器            */
18 #define AP3216C_INTSTATUS   0X01          /* 中断状态寄存器        */
19 #define AP3216C_INTCLEAR    0X02          /* 中断清除寄存器        */
20 #define AP3216C_IRDATALOW   0x0A          /* IR 数据低字节          */
21 #define AP3216C_IRDATAHIGH  0x0B          /* IR 数据高字节          */
22 #define AP3216C_ALSDATALOW  0x0C          /* ALS 数据低字节          */
23 #define AP3216C_ALSDATAHIGH 0X0D          /* ALS 数据高字节          */
24 #define AP3216C_PSDATALOW   0X0E          /* PS 数据低字节          */
25 #define AP3216C_PSDATAHIGH  0X0F          /* PS 数据高字节          */
26
27 #endif
    
```

ap3216creg.h 没什么好讲的，就是一些寄存器宏定义。然后在 ap3216c.c 输入如下内容：

示例代码 28.6.2.2 ap3216c.c 文件代码段

```

1  /*****
2  Copyright © ALIENTEK Co., Ltd. 1998-2029. All rights reserved.
3  文件名      : ap3216c.c
4  作者        : 正点原子 Linux 团队
5  版本        : V1.0
6  描述        : AP3216C 驱动程序
7  其他        : 无
8  论坛        : www.openedv.com
9  日志        : 初版 V1.0 2021/03/19 正点原子 Linux 团队创建
10 *****/
11 #include <linux/types.h>
12 #include <linux/kernel.h>
13 #include <linux/delay.h>
14 #include <linux/ide.h>
    
```

```

15 #include <linux/init.h>
16 #include <linux/module.h>
17 #include <linux/errno.h>
18 #include <linux/gpio.h>
19 #include <linux/cdev.h>
20 #include <linux/device.h>
21 #include <linux/of_gpio.h>
22 #include <linux/semaphore.h>
23 #include <linux/timer.h>
24 #include <linux/i2c.h>
25 // #include <asm/mach/map.h>
26 #include <asm/uaccess.h>
27 #include <asm/io.h>
28 #include "ap3216creg.h"
29
30 #define AP3216C_CNT 1
31 #define AP3216C_NAME "ap3216c"
32
33 struct ap3216c_dev {
34     struct i2c_client *client; /* i2c 设备 */
35     dev_t devid; /* 设备号 */
36     struct cdev cdev; /* cdev */
37     struct class *class; /* 类 */
38     struct device *device; /* 设备 */
39     struct device_node *nd; /* 设备节点 */
40     unsigned short ir, als, ps; /* 三个光传感器数据 */
41 };
42
43 /*
44  * @description : 从 ap3216c 读取多个寄存器数据
45  * @param - dev : ap3216c 设备
46  * @param - reg : 要读取的寄存器首地址
47  * @param - val : 读取到的数据
48  * @param - len : 要读取的数据长度
49  * @return : 操作结果
50  */
51 static int ap3216c_read_regs(struct ap3216c_dev *dev, u8 reg,
                             void *val, int len)
52 {
53     int ret;
54     struct i2c_msg msg[2];
55     struct i2c_client *client = (struct i2c_client *)dev->client;
56

```

```

57     /* msg[0]为发送要读取的首地址 */
58     msg[0].addr = client->addr;           /* ap3216c 地址 */
59     msg[0].flags = 0;                    /* 标记为发送数据 */
60     msg[0].buf = &reg;                   /* 读取的首地址 */
61     msg[0].len = 1;                      /* reg 长度 */
62
63     /* msg[1]读取数据 */
64     msg[1].addr = client->addr;           /* ap3216c 地址 */
65     msg[1].flags = I2C_M_RD;             /* 标记为读取数据 */
66     msg[1].buf = val;                    /* 读取数据缓冲区 */
67     msg[1].len = len;                    /* 要读取的数据长度 */
68
69     ret = i2c_transfer(client->adapter, msg, 2);
70     if(ret == 2) {
71         ret = 0;
72     } else {
73         printk("i2c rd failed=%d reg=%06x len=%d\n",ret, reg, len);
74         ret = -EREMOTEIO;
75     }
76     return ret;
77 }
78
79 /*
80  * @description: 向 ap3216c 多个寄存器写入数据
81  * @param - dev: ap3216c 设备
82  * @param - reg: 要写入的寄存器首地址
83  * @param - val: 要写入的数据缓冲区
84  * @param - len: 要写入的数据长度
85  * @return   : 操作结果
86  */
87 static s32 ap3216c_write_regs(struct ap3216c_dev *dev, u8 reg,
88                               u8 *buf, u8 len)
89 {
90     u8 b[256];
91     struct i2c_msg msg;
92     struct i2c_client *client = (struct i2c_client *)dev->client;
93
94     b[0] = reg;                           /* 寄存器首地址 */
95     memcpy(&b[1],buf,len);                /* 将要写入的数据拷贝到数组 b 里面 */
96
97     msg.addr = client->addr;               /* ap3216c 地址 */
98     msg.flags = 0;                        /* 标记为写数据 */

```

```

99     msg.buf = b;                               /* 要写入的数据缓冲区 */
100    msg.len = len + 1;                          /* 要写入的数据长度 */
101
102    return i2c_transfer(client->adapter, &msg, 1);
103 }
104
105 /*
106 * @description: 读取 ap3216c 指定寄存器值, 读取一个寄存器
107 * @param - dev: ap3216c 设备
108 * @param - reg: 要读取的寄存器
109 * @return   :   读取到的寄存器值
110 */
111 static unsigned char ap3216c_read_reg(struct ap3216c_dev *dev,
                                       u8 reg)
112 {
113     u8 data = 0;
114
115     ap3216c_read_regs(dev, reg, &data, 1);
116     return data;
117 }
118
119 /*
120 * @description: 向 ap3216c 指定寄存器写入指定的值, 写一个寄存器
121 * @param - dev: ap3216c 设备
122 * @param - reg: 要写的寄存器
123 * @param - data: 要写入的值
124 * @return   :   无
125 */
126 static void ap3216c_write_reg(struct ap3216c_dev *dev, u8 reg,
                                u8 data)
127 {
128     u8 buf = 0;
129     buf = data;
130     ap3216c_write_regs(dev, reg, &buf, 1);
131 }
132
133 /*
134 * @description   : 读取 AP3216C 的数据, 包括 ALS, PS 和 IR, 注意! 如果同时
135 *                 : 打开 ALS, IR+PS 两次数据读取的时间间隔要大于 112.5ms
136 * @param - ir    : ir 数据
137 * @param - ps    : ps 数据
138 * @param - ps    : als 数据
139 * @return        : 无。

```

```

140 */
141 void ap3216c_readdata(struct ap3216c_dev *dev)
142 {
143     unsigned char i = 0;
144     unsigned char buf[6];
145
146     /* 循环读取所有传感器数据 */
147     for(i = 0; i < 6; i++) {
148         buf[i] = ap3216c_read_reg(dev, AP3216C_IRDATALOW + i);
149     }
150
151     if(buf[0] & 0X80) /* IR_OF 位为 1, 则数据无效 */
152         dev->ir = 0;
153     else /* 读取 IR 传感器的数据 */
154         dev->ir = ((unsigned short)buf[1] << 2) | (buf[0] & 0X03);
155
156     dev->als = ((unsigned short)buf[3] << 8) | buf[2];
157
158     if(buf[4] & 0x40) /* IR_OF 位为 1, 则数据无效 */
159         dev->ps = 0;
160     else /* 读取 PS 传感器的数据 */
161         dev->ps = ((unsigned short)(buf[5] & 0X3F) << 4) | (buf[4] &
162 0X0F);
163
164 /*
165 * @description : 打开设备
166 * @param - inode: 传递给驱动的 inode
167 * @param - filp : 设备文件, file 结构体有个叫做 private_data 的成员变量
168 * 一般在 open 的时候将 private_data 指向设备结构体。
169 * @return : 0 成功; 其他 失败
170 */
171 static int ap3216c_open(struct inode *inode, struct file *filp)
172 {
173     /* 从 file 结构体获取 cdev 指针, 再根据 cdev 获取 ap3216c_dev 首地址 */
174     struct cdev *cdev = filp->f_path.dentry->d_inode->i_cdev;
175     struct ap3216c_dev *ap3216cdev = container_of(cdev,
176 struct ap3216c_dev, cdev);
177
178     /* 初始化 AP3216C */
179     ap3216c_write_reg(ap3216cdev, AP3216C_SYSTEMCONG, 0x04);
180     mdelay(50);
181     ap3216c_write_reg(ap3216cdev, AP3216C_SYSTEMCONG, 0X03);

```

```

181     return 0;
182 }
183
184 /*
185  * @description   : 从设备读取数据
186  * @param - filp  : 要打开的设备文件(文件描述符)
187  * @param - buf   : 返回给用户空间的数据缓冲区
188  * @param - cnt   : 要读取的数据长度
189  * @param - offt  : 相对于文件首地址的偏移
190  * @return        : 读取的字节数, 如果为负值, 表示读取失败
191  */
192 static ssize_t ap3216c_read(struct file *filp, char __user *buf,
                             size_t cnt, loff_t *off)
193 {
194     short data[3];
195     long err = 0;
196
197     struct cdev *cdev = filp->f_path.dentry->d_inode->i_cdev;
198     struct ap3216c_dev *dev = container_of(cdev, struct ap3216c_dev,
                                             cdev);
199
200     ap3216c_readdata(dev);
201
202     data[0] = dev->ir;
203     data[1] = dev->als;
204     data[2] = dev->ps;
205     err = copy_to_user(buf, data, sizeof(data));
206     return 0;
207 }
208
209 /*
210  * @description   : 关闭/释放设备
211  * @param - filp  : 要关闭的设备文件(文件描述符)
212  * @return        : 0 成功;其他 失败
213  */
214 static int ap3216c_release(struct inode *inode, struct file *filp)
215 {
216     return 0;
217 }
218
219 /* AP3216C 操作函数 */
220 static const struct file_operations ap3216c_ops = {
221     .owner = THIS_MODULE,

```

```

222     .open = ap3216c_open,
223     .read = ap3216c_read,
224     .release = ap3216c_release,
225 };
226
227 /*
228  * @description : i2c 驱动的 probe 函数, 当驱动与
229  *               设备匹配以后此函数就会执行
230  * @param - client : i2c 设备
231  * @param - id     : i2c 设备 ID
232  * @return        : 0, 成功; 其他负值, 失败
233  */
234 static int ap3216c_probe(struct i2c_client *client,
                          const struct i2c_device_id *id)
235 {
236     int ret;
237     struct ap3216c_dev *ap3216cdev;
238
239
240     ap3216cdev = devm_kzalloc(&client->dev, sizeof(*ap3216cdev),
                              GFP_KERNEL);
241
242     if(!ap3216cdev)
243         return -ENOMEM;
244
245     /* 注册字符设备驱动 */
246     /* 1、创建设备号 */
247     ret = alloc_chrdev_region(&ap3216cdev->devid, 0, AP3216C_CNT,
                              AP3216C_NAME);
248
249     if(ret < 0) {
250         pr_err("%s Couldn't alloc_chrdev_region, ret=%d\r\n",
                AP3216C_NAME, ret);
251         return -ENOMEM;
252     }
253
254     /* 2、初始化 cdev */
255     ap3216cdev->cdev.owner = THIS_MODULE;
256     cdev_init(&ap3216cdev->cdev, &ap3216c_ops);
257
258     /* 3、添加一个 cdev */
259     ret = cdev_add(&ap3216cdev->cdev, ap3216cdev->devid,
                   AP3216C_CNT);
260
261     if(ret < 0) {
262         goto del_unregister;
263     }
264 }

```

```

260     }
261
262     /* 4、创建类 */
263     ap3216cdev->class = class_create(THIS_MODULE, AP3216C_NAME);
264     if (IS_ERR(ap3216cdev->class)) {
265         goto del_cdev;
266     }
267
268     /* 5、创建设备 */
269     ap3216cdev->device = device_create(ap3216cdev->class, NULL,
270                                     ap3216cdev->devid, NULL, AP3216C_NAME);
271     if (IS_ERR(ap3216cdev->device)) {
272         goto destroy_class;
273     }
274     ap3216cdev->client = client;
275     /* 保存 ap3216cdev 结构体 */
276     i2c_set_clientdata(client, ap3216cdev);
277
278     return 0;
279 destroy_class:
280     device_destroy(ap3216cdev->class, ap3216cdev->devid);
281 del_cdev:
282     cdev_del(&ap3216cdev->cdev);
283 del_unregister:
284     unregister_chrdev_region(ap3216cdev->devid, AP3216C_CNT);
285     return -EIO;
286 }
287 /*
288  * @description : i2c 驱动的 remove 函数, 移除 i2c 驱动的时候此函数会执行
289  * @param - client : i2c 设备
290  * @return      : 0, 成功;其他负值, 失败
291  */
292 static int ap3216c_remove(struct i2c_client *client)
293 {
294     struct ap3216c_dev *ap3216cdev = i2c_get_clientdata(client);
295     /* 注销字符设备驱动 */
296     /* 1、删除 cdev */
297     cdev_del(&ap3216cdev->cdev);
298     /* 2、注销设备号 */
299     unregister_chrdev_region(ap3216cdev->devid, AP3216C_CNT);
300     /* 3、注销设备 */
301     device_destroy(ap3216cdev->class, ap3216cdev->devid);

```

```

302     /* 4、注销类 */
303     class_destroy(ap3216cdev->class);
304     return 0;
305 }
306
307 /* 传统匹配方式 ID 列表 */
308 static const struct i2c_device_id ap3216c_id[] = {
309     {"alientek,ap3216c", 0},
310     {}
311 };
312
313 /* 设备树匹配列表 */
314 static const struct of_device_id ap3216c_of_match[] = {
315     { .compatible = "alientek,ap3216c" },
316     { /* Sentinel */ }
317 };
318
319 /* i2c 驱动结构体 */
320 static struct i2c_driver ap3216c_driver = {
321     .probe = ap3216c_probe,
322     .remove = ap3216c_remove,
323     .driver = {
324         .owner = THIS_MODULE,
325         .name = "ap3216c",
326         .of_match_table = ap3216c_of_match,
327     },
328     .id_table = ap3216c_id,
329 };
330
331 /*
332  * @description   : 驱动入口函数
333  * @param         : 无
334  * @return        : 无
335  */
336 static int __init ap3216c_init(void)
337 {
338     int ret = 0;
339
340     ret = i2c_add_driver(&ap3216c_driver);
341     return ret;
342 }
343
344 /*

```

```

345 * @description   : 驱动出口函数
346 * @param        : 无
347 * @return       : 无
348 */
349 static void __exit ap3216c_exit(void)
350 {
351     i2c_del_driver(&ap3216c_driver);
352 }
353
354 /* module_i2c_driver(ap3216c_driver) */
355
356 module_init(ap3216c_init);
357 module_exit(ap3216c_exit);
358 MODULE_LICENSE("GPL");
359 MODULE_AUTHOR("ALIENTEK");
360 MODULE_INFO(intree, "Y");
    
```

在示例代码 28.6.2.2 里，没有定义一个全局变量，那是因为 linux 内核不推荐使用全局变量，要使用内存的就用 `devm_kzalloc` 之类的函数去申请空间。

第 33~41 行，自定义一个 `ap3216c_dev` 结构体。第 34 行的 `client` 成员变量用来存储从设备树提供的 `i2c_client` 结构体。第 40 行的 `ir`、`als` 和 `ps` 分别存储 AP3216C 的 IR、ALS 和 PS 数据。

第 51~77 行，`ap3216c_read_regs` 函数实现多字节读取，但是 AP3216C 好像不支持连续多字节读取，此函数在测试其他 I2C 设备的时候可以实现多给字节连续读取，但是在 AP3216C 上不能连续读取多个字节，不过读取一个字节没有问题的。

第 87~103 行，`ap3216c_write_regs` 函数实现连续多字节写操作。

第 111~117 行，`ap3216c_read_reg` 函数用于读取 AP3216C 的指定寄存器数据，用于一个寄存器的数据读取。

第 126~131 行，`ap3216c_write_reg` 函数用于向 AP3216C 的指定寄存器写入数据，用于一个寄存器的数据写操作。

第 141~162 行，读取 AP3216C 的 PS、ALS 和 IR 等传感器原始数据值。

第 171~225 行，标准的字符设备驱动框架。`ap3216c_dev` 结构体里有一个 `cdev` 的变量成员，第 174 行就是获取 `ap3216c_dev` 里的 `cdev` 这个变量的地址，在第 175 行使用 `container_of` 宏获取 `ap3216c_dev` 的首地址。

第 234~285 行，`ap3216c_probe` 函数，当 I2C 设备和驱动匹配成功以后此函数就会执行，和 `platform` 驱动框架一样。此函数前面都是标准的字符设备注册代码，第 275 行，调用 `i2c_set_clientdata` 函数将 `ap3216cdev` 变量的地址绑定到 `client`，进行绑定之后，可以通过 `i2c_get_clientdata` 来获取 `ap3216cdev` 变量指针。

第 292~305 行，`ap3216c_remove` 函数，当 I2C 驱动模块卸载时会执行此函数。第 294 行通过调用 `i2c_get_clientdata` 函数来得到 `ap3216cdev` 变量的地址，后面执行的一系列卸载、注销操作都是前面讲到过的标准字符设备。

第 308~311 行，`ap3216c_id` 匹配表，`i2c_device_id` 类型。用于传统的设备和驱动匹配，也就是没有使用设备树的时候。

第 314~317 行, ap3216c_of_match 匹配表, of_device_id 类型, 用于设备树设备和驱动匹配。这里只写了一个 compatible 属性, 值为“alientek,ap3216c”。

第 320~329 行, ap3216c_driver 结构体变量, i2c_driver 类型。

第 336~342 行, 驱动入口函数 ap3216c_init, 此函数通过调用 i2c_add_driver 来向 Linux 内核注册 i2c_driver, 也就是 ap3216c_driver。

第 349~352 行, 驱动出口函数 ap3216c_exit, 此函数通过调用 i2c_del_driver 来注销掉前面注册的 ap3216c_driver。

28.6.3 编写测试 APP

新建 ap3216cApp.c 文件, 然后在里面输入如下所示内容:

示例代码 28.6.3.1 测试 APP

```

12 #include "stdio.h"
13 #include "unistd.h"
14 #include "sys/types.h"
15 #include "sys/stat.h"
16 #include "sys/ioctl.h"
17 #include "fcntl.h"
18 #include "stdlib.h"
19 #include "string.h"
20 #include <poll.h>
21 #include <sys/select.h>
22 #include <sys/time.h>
23 #include <signal.h>
24 #include <fcntl.h>
25 /*
26  * @description   : main 主程序
27  * @param - argc  : argv 数组元素个数
28  * @param - argv  : 具体参数
29  * @return        : 0 成功;其他 失败
30  */
31 int main(int argc, char *argv[])
32 {
33     int fd;
34     char *filename;
35     unsigned short databuf[3];
36     unsigned short ir, als, ps;
37     int ret = 0;
38
39     if (argc != 2) {
40         printf("Error Usage!\r\n");
41         return -1;
42     }
43

```

```

44     filename = argv[1];
45     fd = open(filename, O_RDWR);
46     if(fd < 0) {
47         printf("can't open file %s\r\n", filename);
48         return -1;
49     }
50
51     while (1) {
52         ret = read(fd, databuf, sizeof(databuf));
53         if(ret == 0) {           /* 数据读取成功 */
54             ir = databuf[0];    /* ir 传感器数据 */
55             als = databuf[1];  /* als 传感器数据 */
56             ps = databuf[2];   /* ps 传感器数据 */
57             printf("ir = %d, als = %d, ps = %d\r\n", ir, als, ps);
58         }
59         usleep(200000);        /*100ms */
60     }
61     close(fd);                 /* 关闭文件 */
62     return 0;
63 }
    
```

ap3216cApp.c 文件内容很简单，就是在 while 循环中不断的读取 AP3216C 的设备文件，从而得到 ir、als 和 ps 这三个数据值，然后将其输出到终端上。

28.7 运行测试

28.7.1 编译驱动程序和测试 APP

1、编译驱动程序

编写 Makefile 文件，本章实验的 Makefile 文件和第五章实验基本一样，只是将 obj-m 变量的值改为“ap3216c.o”，Makefile 内容如下所示：

示例代码 28.7.1.1 Makefile 文件

```

1  KERNELDIR := /home/alientek/rk3568_linux_sdk/kernel
.....
4  obj-m := ap3216c.o
.....
11 clean:
12 $(MAKE) -C $(KERNELDIR) M=$(CURRENT_PATH) clean
    
```

第 4 行，设置 obj-m 变量的值为“ap3216c.o”。

输入如下命令编译出驱动模块文件：

```
make ARCH=arm64 //ARCH=arm64 必须指定，否则编译会失败
编译成功以后就会生成一个名为“ap3216c.ko”的驱动模块文件。
```

2、编译测试 APP

输入如下命令编译 ap3216cApp.c 这个测试程序：

```
/opt/atk-dlrk356x-toolchain/bin/aarch64-buildroot-linux-gnu-gcc ap3216cApp.c -o ap3216cApp
```

编译成功以后就会生成 ap3216cApp 这个应用程序。

28.7.2 运行测试

在 Ubuntu 中将上一小节编译出来的 app3216c.ko 和 ap3216cApp 通过 adb 命令发送到开发板的 /lib/modules/4.19.232 目录下，命令如下：

```
adb push ap3216c.ko ap3216cApp /lib/modules/4.19.232
```

加载驱动。

```
depmod //第一次加载驱动的时候需要运行此命令
```

```
modprobe ap3216c //加载驱动模块
```

当驱动模块加载成功以后使用 ap3216cApp 来测试，输入如下命令：

```
./ap3216cApp /dev/ap3216c
```

测试 APP 会不断的从 AP3216C 中读取数据，然后输出到终端上，如图 28.7.2.1 所示：

```
root@ATK-DLRK356X:/lib/modules/4.19.232# ./ap3216cApp /dev/ap3216c
ir = 0, als = 0, ps = 0
ir = 13, als = 330, ps = 0
ir = 0, als = 322, ps = 16
ir = 7, als = 322, ps = 9
ir = 13, als = 330, ps = 0
ir = 11, als = 325, ps = 5
ir = 4, als = 328, ps = 0
ir = 16, als = 327, ps = 0
ir = 11, als = 142, ps = 65
ir = 18, als = 85, ps = 47
```

图 28.7.2.1 获取到的 AP3216C 数据

大家可以用手电筒照一下 AP3216C，或者手指靠近 AP3216C 来观察传感器数据有没有变化。

第二十九章 Linux RTC 驱动实验

RTC 也就是实时时钟，用于记录当前系统时间，对于 Linux 系统而言时间是非常重要的，就和我们使用 Windows 电脑或手机查看时间一样，我们在使用 Linux 设备的时候也需要查看时间。本章我们就来学习一下如何编写 Linux 下的 RTC 驱动程序。

29.1 Linux 内核 RTC 驱动简介

RTC 设备驱动是一个标准的字符设备驱动,应用程序通过 `open`、`release`、`read`、`write` 和 `ioctl` 等函数完成对 RTC 设备的操作,本章我们主要学习如何使用 RK3568 核心板上的 RK809 自带的 RTC 外设。

Linux 内核将 RTC 设备抽象为 `rtc_device` 结构体,因此 RTC 设备驱动就是申请并初始化 `rtc_device`,最后将 `rtc_device` 注册到 Linux 内核里面,这样 Linux 内核就有一个 RTC 设备的。至于 RTC 设备的操作肯定是用一个操作集合(结构体)来表示的,我们先来看一下 `rtc_device` 结构体,此结构体定义在 `include/linux/rtc.h` 文件中,结构体内容如下(删除条件编译):

示例代码 29.1.1 `rtc_device` 结构体

```
100 struct rtc_device {
101     struct device dev;          /* 设备 */
102     struct module *owner;
103
104     int id;                     /* ID */
105
106     const struct rtc_class_ops *ops; /* RTC 设备底层操作函数 */
107     struct mutex ops_lock;
108
109     struct cdev char_dev;      /* 字符设备 */
110     unsigned long flags;
111
112     unsigned long irq_data;
113     spinlock_t irq_lock;
114     wait_queue_head_t irq_queue;
115     struct fasync_struct *async_queue;
116
117     int irq_freq;
118     int max_user_freq;
119
120     struct timerqueue_head timerqueue;
121     struct rtc_timer aie_timer;
122     struct rtc_timer uie_rtctimer;
123     struct hrtimer pie_timer; /* sub second exp, so needs hrtimer */
124     int pie_enabled;
125     struct work_struct irqwork;
126     /* Some hardware can't support UIE mode */
127     int uie_unsupported;
128     .....
129 };
```

我们需要重点关注的是 `ops` 成员变量,这是一个 `rtc_class_ops` 类型的指针变量,`rtc_class_ops` 为 RTC 设备的最底层操作函数集合,包括从 RTC 设备中读取时间、向 RTC 设备写入新的时间值等。因此,`rtc_class_ops` 是需要用户根据所使用的 RTC 设备编写的,此结构体定义在

include/linux/rtc.h 文件中, 内容如下:

示例代码 29.1.2 rtc_class_ops 结构体

```

75 struct rtc_class_ops {
76     int (*ioctl)(struct device *, unsigned int, unsigned long);
77     int (*read_time)(struct device *, struct rtc_time *);
78     int (*set_time)(struct device *, struct rtc_time *);
79     int (*read_alarm)(struct device *, struct rtc_wkalrm *);
80     int (*set_alarm)(struct device *, struct rtc_wkalrm *);
81     int (*proc)(struct device *, struct seq_file *);
82     int (*alarm_irq_enable)(struct device *, unsigned int enabled);
83     int (*read_offset)(struct device *, long *offset);
84     int (*set_offset)(struct device *, long offset);
85 };
    
```

看名字就知道 `rtc_class_ops` 操作集合中的这些函数是做什么的了, 但是我们要注意, `rtc_class_ops` 中的这些函数只是最底层的 RTC 设备操作函数, 并不是提供给应用层的 `file_operations` 函数操作集。RTC 是个字符设备, 那么肯定有字符设备的 `file_operations` 函数操作集, Linux 内核提供了一个 RTC 通用字符设备驱动文件, 文件名为 `drivers/rtc/rtc-dev.c`, `rtc-dev.c` 文件提供了所有 RTC 设备共用的 `file_operations` 函数操作集, 如下所示:

示例代码 29.1.3 RTC 通用 file_operations 操作集

```

431 static const struct file_operations rtc_dev_fops = {
432     .owner      = THIS_MODULE,
433     .llseek    = no_llseek,
434     .read      = rtc_dev_read,
435     .poll      = rtc_dev_poll,
436     .unlocked_ioctl = rtc_dev_ioctl,
437     .open      = rtc_dev_open,
438     .release   = rtc_dev_release,
439     .fsync     = rtc_dev_fsync,
440 };
    
```

看到示例代码 29.1.3 是不是很熟悉了, 标准的字符设备操作集。应用程序可以通过 `ioctl` 函数来设置/读取时间、设置/读取闹钟的操作, 对应的 `rtc_dev_ioctl` 函数就会执行。 `rtc_dev_ioctl` 最终会通过操作 `rtc_class_ops` 中的 `read_time`、`set_time` 等函数来对具体 RTC 设备的读写操作。我们简单来看一下 `rtc_dev_ioctl` 函数, 函数内容如下(有省略):

示例代码 29.1.4 rtc_dev_ioctl 函数代码段

```

202 static long rtc_dev_ioctl(struct file *file,
203                          unsigned int cmd, unsigned long arg)
204 {
205     int err = 0;
206     struct rtc_device *rtc = file->private_data;
207     const struct rtc_class_ops *ops = rtc->ops;
208     struct rtc_time tm;
209     struct rtc_wkalrm alarm;
210     void __user *uarg = (void __user *)arg;
    
```

```

211
212     err = mutex_lock_interruptible(&rtc->ops_lock);
213     if (err)
214         return err;
.....
253     switch (cmd) {
.....
317     case RTC_RD_TIME:        /* 读取时间 */
318         mutex_unlock(&rtc->ops_lock);
319
320         err = rtc_read_time(rtc, &tm);
321         if (err < 0)
322             return err;
323
324         if (copy_to_user(uarg, &tm, sizeof(tm)))
325             err = -EFAULT;
326         return err;
327
328     case RTC_SET_TIME:       /* 设置时间 */
329         mutex_unlock(&rtc->ops_lock);
330
331         if (copy_from_user(&tm, uarg, sizeof(tm)))
332             return -EFAULT;
333
334         return rtc_set_time(rtc, &tm);
.....
385     default:
386         /* Finally try the driver's ioctl interface */
387         if (ops->ioctl) {
388             err = ops->ioctl(rtc->dev.parent, cmd, arg);
389             if (err == -ENOIOCTLCMD)
390                 err = -ENOTTY;
391         } else {
392             err = -ENOTTY;
393         }
394         break;
395     }
396
397 done:
398     mutex_unlock(&rtc->ops_lock);
399     return err;
400 }
    
```

第 317 行, RTC_RD_TIME 为时间读取命令。

第 320 行, 如果是读取时间命令的话就调用 `rtc_read_time` 函数获取当前 RTC 时钟, `rtc_read_time` 会调用 `__rtc_read_time` 函数, `__rtc_read_time` 函数内容如下:

示例代码 29.1.5 `__rtc_read_time` 函数代码段

```

84 static int __rtc_read_time(struct rtc_device *rtc, struct rtc_time *tm)
85 {
86     int err;
87
88     if (!rtc->ops) {
89         err = -ENODEV;
90     } else if (!rtc->ops->read_time) {
91         err = -EINVAL;
92     } else {
93         memset(tm, 0, sizeof(struct rtc_time));
94         err = rtc->ops->read_time(rtc->dev.parent, tm);
95         if (err < 0) {
96             dev_dbg(&rtc->dev, "read_time: fail to read: %d\n",
97                 err);
98             return err;
99         }
100
101         rtc_add_offset(rtc, tm);
102
103         err = rtc_valid_tm(tm);
104         if (err < 0)
105             dev_dbg(&rtc->dev, "read_time: rtc_time isn't valid\n");
106     }
107     return err;
108 }

```

从第 94 行可以看出, `__rtc_read_time` 函数会通过调用 `rtc_class_ops` 中的 `read_time` 成员变量来从 RTC 设备中获取当前时间。`rtc_dev_ioctl` 函数对其他的命令处理都是类似的, 比如 `RTC_ALM_READ` 命令会通过 `rtc_read_alarm` 函数获取到闹钟值, 而 `rtc_read_alarm` 函数经过层层调用, 最终会调用 `rtc_class_ops` 中的 `read_alarm` 函数来获取闹钟值。

至此, Linux 内核中 RTC 驱动调用流程就很清晰了, 如图 29.1.1 所示:

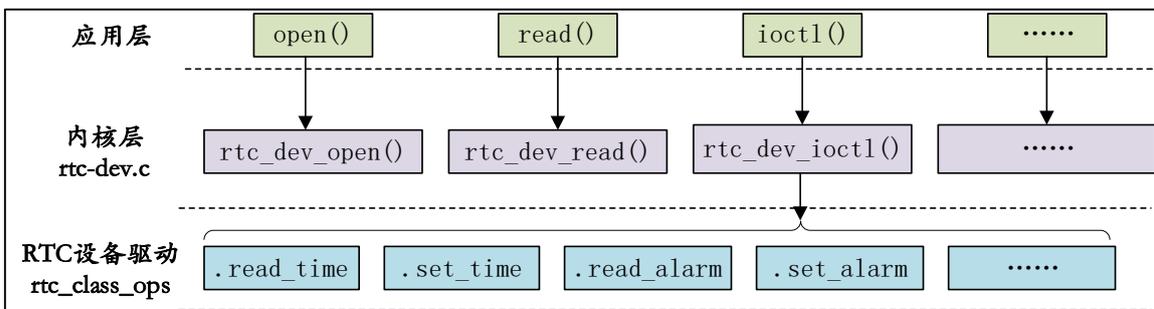


图 29.1.1 Linux RTC 驱动调用流程

当 `rtc_class_ops` 准备好以后需要将其注册到 Linux 内核中, 这里我们可以使用

rtc_device_register 函数完成注册工作。此函数会申请一个 rtc_device 并且初始化这个 rtc_device, 最后向调用者返回这个 rtc_device, 此函数原型如下:

```
struct rtc_device *rtc_device_register(const char      *name,
                                       struct device   *dev,
                                       const struct rtc_class_ops *ops,
                                       struct module   *owner)
```

函数参数和返回值含义如下:

name: 设备名字。

dev: 设备。

ops: RTC 底层驱动函数集。

owner: 驱动模块拥有者。

返回值: 注册成功的话就返回 rtc_device, 错误的话会返回一个负值。

当卸载 RTC 驱动的时候需要调用 rtc_device_unregister 函数来注销注册的 rtc_device, 函数原型如下:

```
void rtc_device_unregister(struct rtc_device *rtc)
```

函数参数和返回值含义如下:

rtc: 要删除的 rtc_device。

返回值: 无。

还有另外一对 rtc_device 注册函数 devm_rtc_device_register 和 devm_rtc_device_unregister, 分别为注册和注销 rtc_device。

29.2 ATK-DLRK3568 核心板 RTC 驱动分析

先直接告诉大家, RK3568 的 RTC 驱动我们不用自己编写, 因为 RK 已经写好了。其实对于大多数的 SOC 来讲, 内部 RTC 驱动都不需要我们去编写, 半导体厂商会编写好。但是这不代表我们就偷懒了, 虽然不用编写 RTC 驱动, 但是我们得看一下这些原厂是怎么编写 RTC 驱动的。

ATK-DLRK3568 核心板有一个电源管理芯片 RK809 挂载在 i2c0 上。RK809 是一款高性能 PMIC, RK809 集成 5 个大电流 DCDC、9 个 LDO、2 个开关 SWITCH、1 个 RTC、1 个高性能 CODEC、可调上电时序等功能。

分析驱动, 先从设备树入手, 打开 arch/arm64/boot/dts/rockchip/rk3568-evb.dtsi, 在里面找到如下 rtc 设备节点, 节点内容如下所示:

示例代码 29.2.1 rk3568-evb.dtsi 文件 pmic 节点

```
1121 rk809: pmic@20 {
1122     compatible = "rockchip,rk809";
1123     reg = <0x20>;
1124     interrupt-parent = <&gpio0>;
1125     interrupts = <3 IRQ_TYPE_LEVEL_LOW>;
1126
1127     pinctrl-names = "default", "pmic-sleep",
1128                  "pmic-power-off", "pmic-reset";
1129     pinctrl-0 = <&pmic_int>;
1130     pinctrl-1 = <&soc_slppin_slp>, <&rk817_slppin_slp>;
1131     pinctrl-2 = <&soc_slppin_gpio>, <&rk817_slppin_pwrn>;
```

```

1132     pinctrl-3 = <&soc_slppin_gpio>, <&rk817_slppin_rst>;
1133
1134     rockchip,system-power-controller;
1135     wakeup-source;
1136     #clock-cells = <1>;
1137     clock-output-names = "rk808-clkout1", "rk808-clkout2";
1138     //fb-inner-reg-idxs = <2>;
1139     /* 1: rst regs (default in codes), 0: rst the pmic */
1140     pmic-reset-func = <0>;
1141     /* not save the PMIC_POWER_EN register in uboot */
1142     not-save-power-en = <1>;
1143
1144     vcc1-supply = <&vcc3v3_sys>;
1145     vcc2-supply = <&vcc3v3_sys>;
1146     vcc3-supply = <&vcc3v3_sys>;
1147     vcc4-supply = <&vcc3v3_sys>;
1148     vcc5-supply = <&vcc3v3_sys>;
1149     vcc6-supply = <&vcc3v3_sys>;
1150     .....
1151 };
    
```

第 1122 行设置兼容属性 `compatible` 的值为“rockchip,rk809”，因此在 Linux 内核源码中搜索此字符串即可找到对应的驱动文件，此文件为 `drivers/mfd/rk808.c`，在 `drivers/mfd/rk808.c` 文件中找到如下所示内容：

示例代码 29.2.2 设备 platform 驱动框架

```

1178 static const struct of_device_id rk808_of_match[] = {
1179     { .compatible = "rockchip,rk805" },
1180     { .compatible = "rockchip,rk808" },
1181     { .compatible = "rockchip,rk809" },
1182     { .compatible = "rockchip,rk816" },
1183     { .compatible = "rockchip,rk817" },
1184     { .compatible = "rockchip,rk818" },
1185     { },
1186 };
1187 MODULE_DEVICE_TABLE(of, rk808_of_match);
1188 .....
1571 static struct i2c_driver rk808_i2c_driver = {
1572     .driver = {
1573         .name = "rk808",
1574         .of_match_table = rk808_of_match,
1575         .pm = &rk8xx_pm_ops,
1576     },
1577     .probe = rk808_probe,
1578     .remove = rk808_remove,
    
```

```
1579 };
```

第 1179~1184 行, 设备树 ID 表。第 1181 行, 刚好有一个 compatible 属性和设备树的 pmic 的 compatible 属性值一样, 所以 pmic 设备节点会和此驱动匹配。

第 1571~1579 行, 标准的 platform 驱动框架, 当设备和驱动匹配成功以后 rk808_probe 函数就会执行, 我们来看一下 rk808_probe 函数, 函数内容如下(有省略):

示例代码 29.2.3 rk808_probe 函数代码段

```
1189 static int rk808_probe(struct i2c_client *client,
1190                        const struct i2c_device_id *id)
1191 {
1192     struct device_node *np = client->dev.of_node;
1193     struct rk808 *rk808;
1194     const struct rk808_reg_data *pre_init_reg;
1195     const struct regmap_irq_chip *battery_irq_chip = NULL;
1196     const struct mfd_cell *cells;
1197     unsigned char pmic_id_msb, pmic_id_lsb;
1198     u8 on_source = 0, off_source = 0;
1199     unsigned int on, off;
1200     int pm_off = 0, msb, lsb;
1201     int nr_pre_init_regs;
1202     int nr_cells;
1203     int ret;
1204     int i;
1205     .....
1212
1213     if (of_device_is_compatible(np, "rockchip,rk817") ||
1214         of_device_is_compatible(np, "rockchip,rk809")) {
1215         pmic_id_msb = RK817_ID_MSB;
1216         pmic_id_lsb = RK817_ID_LSB;
1217     } else {
1218         pmic_id_msb = RK808_ID_MSB;
1219         pmic_id_lsb = RK808_ID_LSB;
1220     }
1221
1222     /* Read chip variant */
1223     msb = i2c_smbus_read_byte_data(client, pmic_id_msb);
1224     if (msb < 0) {
1225         dev_err(&client->dev, "failed to read the chip id at
1226         0x%x\n",
1227                 RK808_ID_MSB);
1228         return msb;
1229     }
1230     lsb = i2c_smbus_read_byte_data(client, pmic_id_lsb);
```

```

1231     if (lsb < 0) {
1232         dev_err(&client->dev, "failed to read the chip id at
0x%x\n",
1233             RK808_ID_LSB);
1234         return lsb;
1235     }
1236
1237     rk808->variant = ((msb << 8) | lsb) & RK8XX_ID_MSK;
1238     dev_info(&client->dev, "chip id: 0x%x\n", (unsigned
int)rk808->variant);
1239
1240     switch (rk808->variant) {
1241     case RK805_ID:
1242         rk808->regmap_cfg = &rk805_regmap_config;
1243         rk808->regmap_irq_chip = &rk805_irq_chip;
1244         pre_init_reg = rk805_pre_init_reg;
1245         nr_pre_init_regs = ARRAY_SIZE(rk805_pre_init_reg);
1246         cells = rk805s;
1247         nr_cells = ARRAY_SIZE(rk805s);
1248         on_source = RK805_ON_SOURCE_REG;
1249         off_source = RK805_OFF_SOURCE_REG;
1250         suspend_reg = rk805_suspend_reg;
1251         suspend_reg_num = ARRAY_SIZE(rk805_suspend_reg);
1252         resume_reg = rk805_resume_reg;
1253         resume_reg_num = ARRAY_SIZE(rk805_resume_reg);
1254         rk808->pm_pwroff_fn = rk805_device_shutdown;
1255         rk808->pm_pwroff_prep_fn =
rk805_device_shutdown_prepare;
1256         break;
1257     case RK808_ID:
1258         rk808->regmap_cfg = &rk808_regmap_config;
1259         rk808->regmap_irq_chip = &rk808_irq_chip;
1260         pre_init_reg = rk808_pre_init_reg;
1261         nr_pre_init_regs = ARRAY_SIZE(rk808_pre_init_reg);
1262         cells = rk808s;
1263         nr_cells = ARRAY_SIZE(rk808s);
1264         rk808->pm_pwroff_fn = rk808_device_shutdown;
1265         break;
1266     case RK816_ID:
1267         rk808->regmap_cfg = &rk816_regmap_config;
1268         rk808->regmap_irq_chip = &rk816_irq_chip;
1269         battery_irq_chip = &rk816_battery_irq_chip;
1270         pre_init_reg = rk816_pre_init_reg;
    
```

```

1271     nr_pre_init_regs = ARRAY_SIZE(rk816_pre_init_reg);
1272     cells = rk816s;
1273     nr_cells = ARRAY_SIZE(rk816s);
1274     on_source = RK816_ON_SOURCE_REG;
1275     off_source = RK816_OFF_SOURCE_REG;
1276     suspend_reg = rk816_suspend_reg;
1277     suspend_reg_num = ARRAY_SIZE(rk816_suspend_reg);
1278     resume_reg = rk816_resume_reg;
1279     resume_reg_num = ARRAY_SIZE(rk816_resume_reg);
1280     rk808->pm_pwroff_fn = rk816_device_shutdown;
1281     break;
1282     case RK818_ID:
1283         rk808->regmap_cfg = &rk818_regmap_config;
1284         rk808->regmap_irq_chip = &rk818_irq_chip;
1285         pre_init_reg = rk818_pre_init_reg;
1286         nr_pre_init_regs = ARRAY_SIZE(rk818_pre_init_reg);
1287         cells = rk818s;
1288         nr_cells = ARRAY_SIZE(rk818s);
1289         on_source = RK818_ON_SOURCE_REG;
1290         off_source = RK818_OFF_SOURCE_REG;
1291         suspend_reg = rk818_suspend_reg;
1292         suspend_reg_num = ARRAY_SIZE(rk818_suspend_reg);
1293         resume_reg = rk818_resume_reg;
1294         resume_reg_num = ARRAY_SIZE(rk818_resume_reg);
1295         rk808->pm_pwroff_fn = rk818_device_shutdown;
1296         break;
1297     case RK809_ID:
1298     case RK817_ID:
1299         rk808->regmap_cfg = &rk817_regmap_config;
1300         rk808->regmap_irq_chip = &rk817_irq_chip;
1301         pre_init_reg = rk817_pre_init_reg;
1302         nr_pre_init_regs = ARRAY_SIZE(rk817_pre_init_reg);
1303         cells = rk817s;
1304         nr_cells = ARRAY_SIZE(rk817s);
1305         on_source = RK817_ON_SOURCE_REG;
1306         off_source = RK817_OFF_SOURCE_REG;
1307         rk808->pm_pwroff_prep_fn = rk817_shutdown_prepare;
1308         of_property_prepare_fn = rk817_of_property_prepare;
1309         pinctrl_init = rk817_pinctrl_init;
1310         break;
1311     default:
1312         dev_err(&client->dev, "Unsupported RK8XX ID %lu\n",
1313             rk808->variant);
    
```

```

1314         return -EINVAL;
1315     }
1316
1317     .....
1391
1392     ret = devm_mfd_add_devices(&client->dev,
1393                               PLATFORM_DEVID_NONE,
1394                               cells, nr_cells, NULL, 0,
1395                               regmap_irq_get_domain(rk808->irq_data));
1396     if (ret) {
1397         dev_err(&client->dev, "failed to add MFD
1398 devices %d\n", ret);
1399         goto err_irq;
1400     }
1401     .....
1429 }
    
```

1213~1214 行, 判断 compatible 是否为 “rockchip,rk809” 则设置读取对应的寄存器。

1240 行, 判断读取出来的 ID, 然后做相应的设置。

1297 行, 判断是否匹配 RK809 芯片, 若匹配, RK817S 芯片的不同功能单元 (MFD cell) 那么赋值为 rk817s 静态结构体数组。

1382~1394 行, 这里添加了平台设备, rtc 设备也是在这个 devm_mfd_add_devices 函数里添加的。

rk817s 静态结构体数组内容如下所示:

示例代码 29.2.4 rk817s 结构体数组

```

263 static const struct mfd_cell rk817s[] = {
264     { .name = "rk808-clkout", },
265     { .name = "rk808-regulator", },
266     { .name = "rk817-battery", .of_compatible = "rk817,battery", },
267     { .name = "rk817-charger", .of_compatible = "rk817,charger", },
268     {
269         .name = "rk805-pwrkey",
270         .num_resources = ARRAY_SIZE(rk817_pwrkey_resources),
271         .resources = &rk817_pwrkey_resources[0],
272     },
273     {
274         .name = "rk808-rtc",
275         .num_resources = ARRAY_SIZE(rk817_rtc_resources),
276         .resources = &rk817_rtc_resources[0],
277     },
278     {
279         .name = "rk817-codec",
280         .of_compatible = "rockchip,rk817-codec",
281     },
    
```

```
282 };
```

通过定义不同的功能单元和相关的资源,可以将这些功能单元配置到 RK817S 芯片的 MFD 子系统中,以实现各种功能,如时钟输出、电源管理、电池管理、充电管理和电源按键等。可以看到第 274 行就有“rk808-rtc”。platform 驱动的名称字段相同,否则的话设备就无法匹配到对应的驱动。

平台设备已经添加了,那么我们要找到“rk808-rtc”的驱动实现。打开 drivers/rtc/rtc-rk808.c 文件,平台设备 name 字段与平台驱动 name 字段匹配,那么 rk808_rtc_probe 才会执行。

示例代码 29.2.5 rk817s 结构体数组

```
512 static struct platform_driver rk808_rtc_driver = {
513     .probe = rk808_rtc_probe,
514     .driver = {
515         .name = "rk808-rtc",
516         .pm = &rk808_rtc_pm_ops,
517     },
518 };
```

rk808_rtc_probe 函数内容如下:

```
417 static int rk808_rtc_probe(struct platform_device *pdev)
418 {
419     struct rk808 *rk808 = dev_get_drvdata(pdev->dev.parent);
420     struct rk808_rtc *rk808_rtc;
421     struct device_node *np;
422     int ret;
423
424     switch (rk808->variant) {
425     case RK805_ID:
426     case RK808_ID:
427     case RK816_ID:
428     case RK818_ID:
429         np = of_get_child_by_name(pdev->dev.parent->of_node,
430             "rtc");
431         if (np && !of_device_is_available(np)) {
432             dev_info(&pdev->dev, "device is disabled\n");
433             return -EINVAL;
434         }
435         break;
436     default:
437         break;
438     }
439
440     rk808_rtc = devm_kzalloc(&pdev->dev, sizeof(*rk808_rtc),
441         GFP_KERNEL);
442     if (rk808_rtc == NULL)
443         return -ENOMEM;
```

```

442
443     switch (rk808->variant) {
444     case RK808_ID:
445     case RK818_ID:
446         rk808_rtc->creg = &rk808_creg;
447         rk808_rtc->flag |= RTC_NEED_TRANSITIONS;
448         break;
449     case RK805_ID:
450     case RK816_ID:
451         rk808_rtc->creg = &rk808_creg;
452         break;
453     case RK809_ID:
454     case RK817_ID:
455         rk808_rtc->creg = &rk817_creg;
456         break;
457     default:
458         rk808_rtc->creg = &rk808_creg;
459         break;
460     }
461     platform_set_drvdata(pdev, rk808_rtc);
462     rk808_rtc->rk808 = rk808;
463
464     /* start rtc running by default, and use shadowed timer. */
465     ret = regmap_update_bits(rk808->regmap,
466                             rk808_rtc->creg->ctrl_reg,
467                             BIT_RTC_CTRL_REG_STOP_RTC_M |
468                             BIT_RTC_CTRL_REG_RTC_READSEL_M,
469                             BIT_RTC_CTRL_REG_RTC_READSEL_M);
470     if (ret) {
471         dev_err(&pdev->dev,
472               "Failed to update RTC control: %d\n", ret);
473         return ret;
474     }
475     ret = regmap_write(rk808->regmap,
476                       rk808_rtc->creg->status_reg,
477                       RTC_STATUS_MASK);
478     if (ret) {
479         dev_err(&pdev->dev,
480               "Failed to write RTC status: %d\n", ret);
481         return ret;
482     }

```

```

483     device_init_wakeup(&pdev->dev, 1);
484
485     rk808_rtc->rtc = devm_rtc_allocate_device(&pdev->dev);
486     if (IS_ERR(rk808_rtc->rtc))
487         return PTR_ERR(rk808_rtc->rtc);
488
489     rk808_rtc->rtc->ops = &rk808_rtc_ops;
490
491     rk808_rtc->irq = platform_get_irq(pdev, 0);
492     if (rk808_rtc->irq < 0) {
493         if (rk808_rtc->irq != -EPROBE_DEFER)
494             dev_err(&pdev->dev, "Wake up is not possible as
495                 irq = %d\n",
496                 rk808_rtc->irq);
497         return rk808_rtc->irq;
498     }
499     /* request alarm irq of rk808 */
500     ret = devm_request_threaded_irq(&pdev->dev, rk808_rtc->irq,
501                                     NULL,
502                                     rk808_alarm_irq, 0,
503                                     "RTC alarm", rk808_rtc);
504     if (ret) {
505         dev_err(&pdev->dev, "Failed to request alarm
506             IRQ %d: %d\n",
507             rk808_rtc->irq, ret);
508         return ret;
509     }
510     return rtc_register_device(rk808_rtc->rtc);
511 }
    
```

第 439 行，调用 `devm_kzalloc` 申请 `rtc` 大小的空间，返回申请空间的首地址。

第 489 行，RTC 底层驱动集为 `rk808_rtc_ops`。`rk808_rtc_ops` 操作集包含了读取/设置 RTC 时间，读取/设置闹钟等函数。

第 500 行，调用 `devm_request_threaded_irq` 函数请求 RTC 中断，中断服务函数为 `rk808_alarm_irq`，用于 RTC 闹钟中断。

第 509 行，调用 `rtc_register_device` 函数向系统注册 `rtc_devcie`。

29.3 RTC 时间查看与设置

29.3.1 使能 RK809 内部 RTC

从上面几节我们可以知道，RK809 内部 RTC 的使能需要先使能 RK809，默认已经使能，我们打开设备树 `rk3568-evb.dtsi`:

示例代码 29.3.1.1 pmic 节点信息

```

1121 rk809: pmic@20 {
1122     compatible = "rockchip,rk809";
1123     reg = <0x20>;
1124     interrupt-parent = <&gpio0>;
1125     interrupts = <3 IRQ_TYPE_LEVEL_LOW>;
1126
1127     pinctrl-names = "default", "pmic-sleep",
1128                   "pmic-power-off", "pmic-reset";
1129     pinctrl-0 = <&pmic_int>;
1130     pinctrl-1 = <&soc_slppin_slp>, <&rk817_slppin_slp>;
1131     pinctrl-2 = <&soc_slppin_gpio>, <&rk817_slppin_pwrnd>;
1132     pinctrl-3 = <&soc_slppin_gpio>, <&rk817_slppin_rst>;
1133
1134     rockchip,system-power-controller;
1135     wakeup-source;
1136     #clock-cells = <1>;
1137     clock-output-names = "rk808-clkout1", "rk808-clkout2";
1138     //fb-inner-reg-idxs = <2>;
1139     /* 1: rst regs (default in codes), 0: rst the pmic */
1140     pmic-reset-func = <0>;
1141     /* not save the PMIC_POWER_EN register in uboot */
1142     not-save-power-en = <1>;
1143
1144     vcc1-supply = <&vcc3v3_sys>;
1145     vcc2-supply = <&vcc3v3_sys>;
1146     vcc3-supply = <&vcc3v3_sys>;
1147     vcc4-supply = <&vcc3v3_sys>;
1148     vcc5-supply = <&vcc3v3_sys>;
1149     vcc6-supply = <&vcc3v3_sys>;
1150     vcc7-supply = <&vcc3v3_sys>;
1151     vcc8-supply = <&vcc3v3_sys>;
1152     vcc9-supply = <&vcc3v3_sys>;
... ..
};
    
```

上面 status 状态没写，默认就是“okay”的。

同时我们需要在 menuconfig 里对应的宏配置为 CONFIG_RTC_DRV_RK808。> Device Drivers > Real Time Clock 选中 CONFIG_RTC_DRV_RK808。如下图。

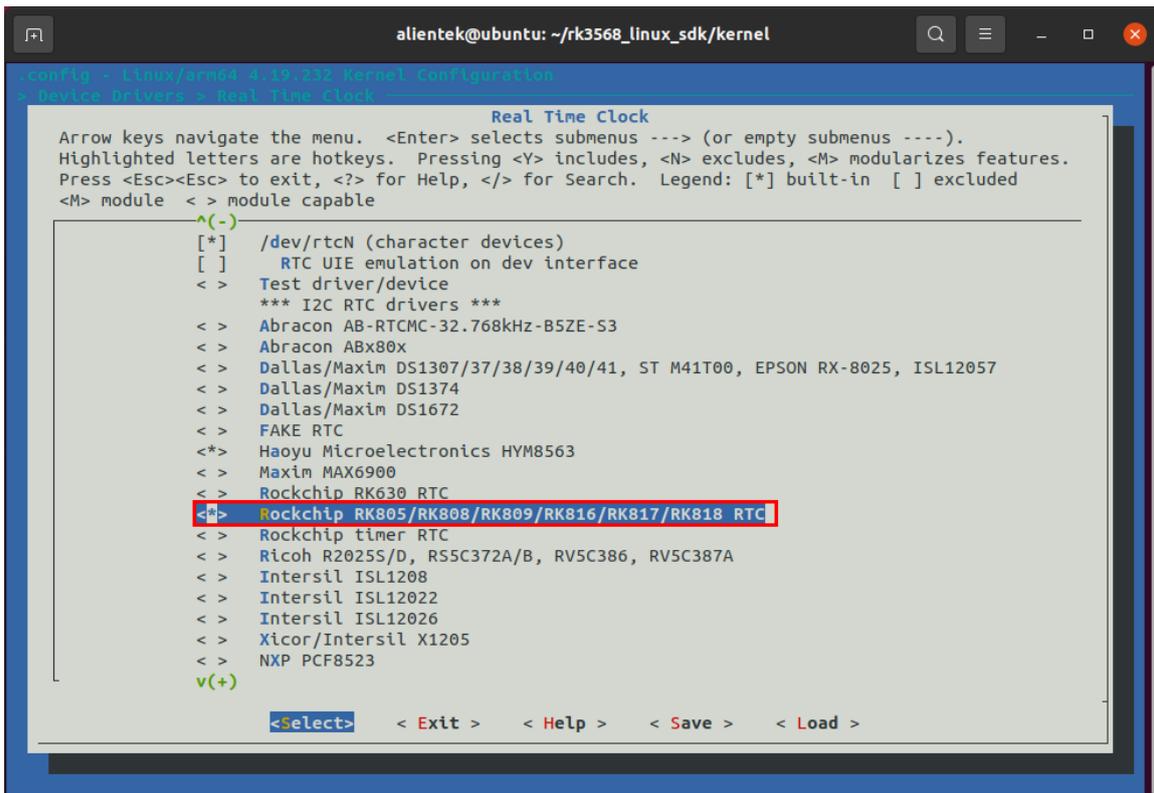


图 29.3.1 RK808 RTC 驱动使能

29.3.2 查看时间

RTC 是用来记时的，因此最基本的就是查看时间，Linux 内核启动的时候可以看到系统时钟设置信息，如图 29.3.2.1 所示：

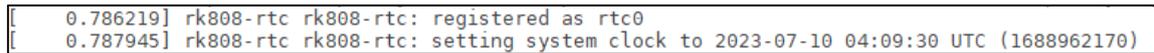


图 29.3.2.1 Linux 启动 log 信息

从图 29.3.2.1 中可以看出可以看到 rk808-rtc 已经注册为 rtc0，有 rtc0 那么可能会有 rtc1，没错 rtc 可以存在多个，比如 ATK-DLRK3568 开发板就有两个 rtc，一个是 pcf8563 rtc 外部时钟芯片。

如果要查看时间的话输入“date”命令即可，结果如图 29.3.2.2 所示：

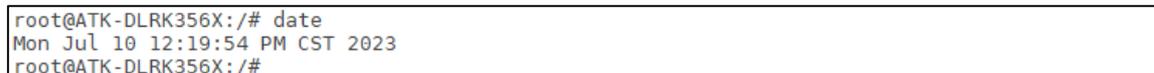


图 29.3.2.2 当前时间值

从上面可看到内核启动 RTC 时间采用的是 UTC 标准，而系统启动后采用的是 CST 标准，时间恰好相差 8 个小时。UTC（协调世界时）和 CST（中部标准时间）是两个不同的时间标准，在中国，CST 通常被解释为“China Standard Time”（中国标准时间），而不是“Central Standard Time”（中部标准时间）。中国 CST 与协调世界时（UTC）相差 8 小时，即 UTC+8。

RTC 时间设置也是使用的 date 命令，输入“date --help”命令即可查看 date 命令如何设置系统时间，结果如图 29.3.2.3 所示：

```

root@ATK-DLRK356X:/# date --help
Usage: date [OPTION]... [+FORMAT]
  or: date [-u|--utc|--universal] [MMDDhhmm[[CC]YY][.ss]]
Display the current time in the given FORMAT, or set the system date.

Mandatory arguments to long options are mandatory for short options too.
-d, --date=STRING      display time described by STRING, not 'now'
--debug                annotate the parsed date,
                       and warn about questionable usage to stderr
-f, --file=DATEFILE   like --date; once for each line of DATEFILE
-I[FMT], --iso-8601[=FMT]
                       output date/time in ISO 8601 format.
                       FMT='date' for date only (the default),
                       'hours', 'minutes', 'seconds', or 'ns'
                       for date and time to the indicated precision.
                       Example: 2006-08-14T02:34:56-06:00
-R, --rfc-email        output date and time in RFC 5322 format.
                       Example: Mon, 14 Aug 2006 02:34:56 -0600
--rfc-3339=FMT         output date/time in RFC 3339 format.
                       FMT='date', 'seconds', or 'ns'
                       for date and time to the indicated precision.
                       Example: 2006-08-14 02:34:56-06:00
-r, --reference=FILE   display the last modification time of FILE
-s, --set=STRING        set time described by STRING
-u, --utc, --universal print or set Coordinated Universal Time (UTC)
--help                 display this help and exit
--version               output version information and exit

FORMAT controls the output.  Interpreted sequences are:

%%      a literal %%
%a      locale's abbreviated weekday name (e.g., Sun)
%A      locale's full weekday name (e.g., Sunday)
%b      locale's abbreviated month name (e.g., Jan)
%B      locale's full month name (e.g., January)
%c      locale's date and time (e.g., Thu Mar 3 23:05:25 2005)
%C      century; like %Y, except omit last two digits (e.g., 20)
%d      day of month (e.g., 01)
%D      date; same as %m/%d/%y
%e      day of month, space padded; same as %_d
%F      full date; same as %Y-%m-%d
%g      last two digits of year of ISO week number (see %G)
%G      year of ISO week number (see %V); normally useful only with %V
%h      same as %b
%H      hour (00..23)
%I      hour (01..12)
%j      day of year (001..366)
%k      hour, space padded ( 0..23); same as %_H
%l      hour, space padded ( 1..12); same as %_I
%m      month (01..12)
%M      minute (00..59)
%n      a newline
%N      nanoseconds (000000000..999999999)
%p      locale's equivalent of either AM or PM; blank if not known
%P      like %p, but lower case
%q      quarter of year (1..4)
%r      locale's 12-hour clock time (e.g., 11:11:04 PM)
%R      24-hour hour and minute; same as %H:%M
%S      seconds since 1970-01-01 00:00:00 UTC
%S      second (00..60)
%t      a tab
%T      time; same as %H:%M:%S
%u      day of week (1..7); 1 is Monday
%U      week number of year, with Sunday as first day of week (00..53)
%V      ISO week number, with Monday as first day of week (01..53)
%w      day of week (0..6); 0 is Sunday
%W      week number of year, with Monday as first day of week (00..53)
%x      locale's date representation (e.g., 12/31/99)
%X      locale's time representation (e.g., 23:13:48)
%y      last two digits of year (00..99)
%Y      year
%z      +hhmm numeric time zone (e.g., -0400)
%:z     +hh:mm numeric time zone (e.g., -04:00)
%::z    +hh:mm:ss numeric time zone (e.g., -04:00:00)
%:::z   numeric time zone with : to necessary precision (e.g., -04, +05:30)
%Z      alphabetic time zone abbreviation (e.g., EDT)

By default, date pads numeric fields with zeroes.
The following optional flags may follow '%':

- (hyphen) do not pad the field
_ (underscore) pad with spaces
0 (zero) pad with zeros
^ use upper case if possible
# use opposite case if possible

After any flags comes an optional field width, as a decimal number;
then an optional modifier, which is either
E to use the locale's alternate representations if available, or
O to use the locale's alternate numeric symbols if available.

Examples:
Convert seconds since the epoch (1970-01-01 UTC) to a date
$ date --date=@2147483647

Show the time on the west coast of the US (use tzselect(1) to find TZ)
$ TZ='America/Los_Angeles' date

Show the local time for 9AM next Friday on the west coast of the US
$ date --date='TZ="America/Los_Angeles" 09:00 next Fri'

GNU coreutils online help: <https://www.gnu.org/software/coreutils/>
Full documentation at: <https://www.gnu.org/software/coreutils/date>
or available locally via: info '(coreutils) date invocation'
root@ATK-DLRK356X:/#
    
```

图 29.3.2.3 date 命令帮助信息

比如现在设置当前时间为 2023 年 7 月 10 日 14:00:00, 因此输入如下命令:

```
date -s "2023-07-10 14:10:00"
```

设置完成以后再次使用 `date` 命令查看一下当前时间就会发现时间改过来了, 如图 29.3.2.4 所示:

```
root@ATK-DLRK356X:/# date
Mon Jul 10 02:10:38 PM CST 2023
root@ATK-DLRK356X:/#
```

图 43.3.2.4 当前时间

大家注意我们使用“`date -s`”命令仅仅是修改了当前时间, 此时间还没有写入到 RK809 内部 RTC 里面或其他的 RTC 芯片里面, 因此系统重启以后时间又会丢失。我们需要将当前的时间写入到 RTC 里面, 这里要用到 `hwclock` 命令, 输入如下命令将系统时间写入到 RTC 里面:

```
hwclock -w //将当前系统时间写入到 RTC 里面
```

时间写入到 RTC 里面以后就不怕系统重启以后时间丢失了, 如果 ATK-DLRK3568 开发板底板接了纽扣电池, 那么开发板即使断电了时间也不会丢失。大家可以尝试一下不断电重启和断电重启这两种情况下开发板时间会不会丢失。(请注意: 由于系统使用了 `ntp` 校时, 如果我们有插网线或者 WIFI 联网, 并且能上网, 那么输入 `date` 查看的时间就是最新的系统时间, RK809 内部硬件时间并不会因 `ntp` 校时而改变。)